Escuela
Politécnica
Superior

Robótica asistencial en lengua de signos

Máster Universitario en Automática y Robótica

Trabajo Fin de Máster

Autor:

Francisco Morillas Espejo

Tutor/es:

Ester Martínez Martín

Angelo Gonçalo Araujo da Silva Costa

Universitat d'Alacant
Universidad de Alicante

Robótica asistencial en lengua de signos

Autor

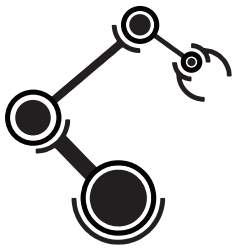
Francisco Morillas Espejo

Tutor/es

Ester Martínez Martín

Angelo Gonçalo Araujo da Silva Costa

Departamento de Ciencia de la Computación e Inteligencia Artificial



Máster Universitario en Automática y Robótica

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Julio 2021

Resumen

El objetivo de este Trabajo Final de Máster es la interpretación de sentencias de la Lengua de Signos Española relativas a tareas diarias mediante técnicas de deep learning con el fin de implementar un intérprete automático. Para el seguimiento de los brazos y manos involucrados en el signado de los diferentes gestos se emplea Openpose, un sistema de detección de cuerpo, cara, pies y manos en tiempo real. Usando una cámara Azure Kinect como herramienta para la captura de imágenes.

Los resultados obtenidos por Openpose son introducidos en una Red Neuronal Artificial donde se produce el reconocimiento. En este trabajo se han estudiado, analizado y comparado distintas arquitecturas con el fin de encontrar la más adecuada para un correcto reconocimiento de signos.

Adicionalmente se abarca la creación de un *dataset* de las diversas palabras de la Lengua de Signos Española necesarias para la creación de dichas frases. Empleando para ello la cámara Azure Kinect y la herramienta Openpose para la extracción de los puntos necesarios en manos y brazos.

Agradecimientos

Agradecer a mi tutora, Ester, por su constante ayuda y guía a lo largo de este trabajo, sin la cual no hubiese sido posible su realización.

Índice general

Índice de figuras

Índice de tablas

1 Introducción

Cerca de un 5% de la población mundial (466 millones de personas) sufre pérdida de audición discapacitante según la Organización Mundial de la Salud (OMS) y se prevé que esta cifra aumente a más de 900 millones de personas (una de cada 10) de aquí a 2050 [8].

Dependiendo del grado de pérdida de audición, el uso de audífonos, implantes cocleares y otros dispositivos pueden mejorar la comunicación. Sin embargo la producción mundial de audífonos satisface menos del 10% de las necesidades mundiales y del 3% en los países en desarrollo [8] lo que obliga a utilizar vías de comunicación alternativas como el texto escrito, la lectura de labios o la lengua de signos.

La lengua de signos es una de las opciones más extendidas, pero aún así plantea inconvenientes como el hecho de que cada país tiene la suya propia y, además, hay variantes dentro de las comunidades o estados de un mismo país. Así pues, no hay una lengua de signos internacional, lo que dificulta aún más la comunicación de este colectivo.

A lo largo de este Trabajo Final de Máster (TFM) se plantea la interpretación de diversas frases de actividades cotidianas obtenidas a partir de un conjunto de palabras de la LSE seleccionadas por su frecuencia de uso. Para ello, se hace uso de Redes Neuronales Artificiales (RNA) de manera que, a partir de una secuencias de imágenes 2D correspondientes al signado de una frase, la RNA sea capaz de interpretarla y traducirla al lenguaje natural.

1.1 Objetivos

El objetivo principal de este TFM consiste en realizar un intérprete basado en secuencias de imágenes 2D mediante RNA, analizando diferentes arquitecturas de dichas redes y con diferentes configuraciones del conjunto de datos a emplear con el fin de determinar la viabilidad del sistema en tiempo real. Para ello, se ha empleado una cámara Azure Kinect [1] como

herramienta para la captura de imágenes.

Con el fin de alcanzar este objetivo principal se han establecido una serie de tareas o subobjetivos que son:

- Estudio de la Lengua de Signos Española. En particular el vocabulario relativo a la alimentación y los utensilios.
- Configuración y control de la cámara Azure Kinect.
- Extracción de las articulaciones involucradas en el signado mediante Openpose.
- Creación de un *dataset* con las diferentes palabras a reconocer.
- Estudio de distintos métodos y/o estrategias para el aprendizaje de dichas palabras.
- Análisis de los resultados obtenidos.

Además, en la Figura 1.1 se muestra un diagrama de flujo con los diferentes pasos llevados a cabo a lo largo del proyecto.

Con los diversos objetivos planteados se pretende tener un sistema funcional para el reconocimiento de ciertas palabras y/o frases dentro de la Lengua de Signos Española empleando la cámara Azure Kinect.

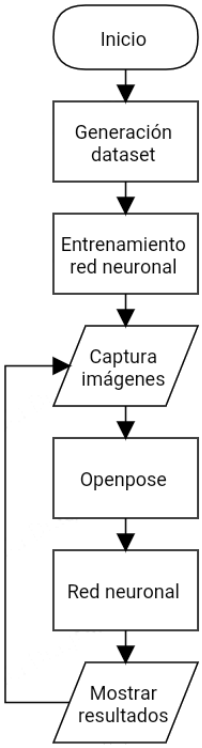


Figura 1.1: Diagrama de flujo del proyecto.

2 Estado de la cuestión

En la literatura diferentes autores han tratado el reconocimiento de signos de diferentes idiomas y desde distintas perspectivas.

Uno de los principales problemas encontrados está relacionado con la adquisición de datos y el tratamiento de éstos. Los primeros sistemas hacían uso de guantes y acelerómetros para capturar datos específicos de las manos. Así, las medidas (x, y, z, orientación, velocidad, etc.) se capturan directamente mediante sensores como el rastreador Polhemos [9] o el DataGlove [10, 11], permitiendo utilizar los datos que proporcionan de manera directa. Sin embargo, estos sistemas plantean distintos inconvenientes como son las restricciones de movimiento, que impiden una gesticulación natural, o el elevado precio de los dispositivos. Esto ha llevado a que las alternativas basadas en visión por computador hayan ganado mucha popularidad en los últimos tiempos.

Desde el punto de la adquisición de datos, la literatura presenta grandes diferencias entre unos autores y otros. Así, Starner *et al.* [12] hacen uso de una cámara frontal junto a una cámara superior enfocada hacia abajo para capturar las manos del signante y, así, mejorar el reconocimiento de los signos. En cambio, [13] usa un sistema de cámaras estéreo para captar la profundidad.

El seguimiento de las manos para el reconocimiento de signos plantea dificultades debido a que, en una conversación normal en Lengua de Signos, la velocidad de las manos es muy rápida además de que las manos interaccionan entre ellas. Esto genera problemas como la oclusión entre manos o la confusión de una mano con la otra. Como solución, algunos autores como Holden y Owens [14] hacen uso de diferentes colores en los dedos para evitar estos problemas.

En el caso de la detección sin guantes ni herramientas especiales, destaca la detección basada en el color de la piel. Un ejemplo de ello es el trabajo presentado por Han *et al.* [15],

que muestra la robustez a oclusiones entre manos y cara mediante el uso de filtros de Kalman para la segmentación de la piel.

En cuanto a la interpretación de los signos, una vez solventado el problema del seguimiento de las manos, se encuentran diferentes métodos. Pigou *et al.* [16] plantean el reconocimiento de la Lengua de Signos Italiana mediante Redes Neuronales Convolucionales (CNN) empleando una cámara Kinect. Para ello proponen una arquitectura consistente en dos CNN, una para la extracción de características de la mano y otra para el cuerpo unidas en la última capa, todo ello concatenado a una RNA compuesta únicamente de Unidades de Rectificado Lineal (ReLUs) con una capa oculta que proporciona la clasificación. Los resultados experimentales obtenidos para el reconocimiento de 20 gestos sobre el conjunto de datos *ChaLearn Looking at People 2014* [17] tienen una precisión del 91.7 % para el conjunto de validación sin importar el signante ni su entorno. Cuando se analizan frases completas es necesario considerar la dimensión temporal. Dentro de esta línea, Fang *et al.* [18] presentan *DeepASL* para la traducción del Lengua de Signos Americana (ASL). En particular, hacen uso de un Leap Motion [19] para extraer las coordenadas 3D de brazos, palmas y dedos, dado que son las partes involucradas en el signado. Este conjunto de características se introduce en una RNN bidireccional de tal manera que su salida se traduce usando Clasificación Temporal Conexionista (CTC). El sistema se ha evaluado con las siguientes palabras de la Lengua de Signos Americana:

- **Pronombres**: *who, I, you, what, we, my, your, other.*
- **Nombres**: *time, food, drink, mother, clothes, box, car, bicycle, book, shoes, year, boy, church, family.*
- **Verbos**: *want, do not want, like, help, finish, need, thank you, meet, live, can, come.*
- **Adjetivos**: *big, small, hot, cold, blue, red, gray, black, green, white, old, with, without, nice, bad, sad, many, sorry, few.*
- **Adverbios**: *where, more, please, but.*

Combinando estas palabras se obtienen 100 frases que se utilizan para la evaluación del sistema, consiguiendo un Top1-WER[1] de un 16.1 $\pm$ 3.7 %. Esto quiere decir que de una frase

[1]Error medio por palabra en la primera predicción.

de cuatro palabras hay una media de 0.64 palabras que requieren de sustitución, inserción o eliminación.

Por otro lado, Starner y Pentland [20] plantean la extracción de la orientación, forma y trayectoria de la mano mediante un filtrado por el color de la piel del usuario. Estos datos son la entrada de un Modelo Oculto de Markov (HMM) encargado del reconocimiento de signos pertenecientes al ASL. Para la fase experimental se han incluido las siguientes palabras:

- **Pronombres**: *I, you, he, we, you (plural), they.*
- **Nombres**: *box, car, book, table, paper, pants, bicycle, bottle, can, wristwatch, umbrella, coat, pencil, shoes, food, magazine, fish, mouse, pill, bowl.*
- **Verbos**: *want, do not want, like, do not like, lose, love, pack, hit, loan.*
- **Adjetivos**: *red, brown, black, gray, yellow.*

3 Herramientas

En este capítulo se presentan las principales características de la cámara Kinect utilizada a lo largo de este proyecto además de las diferentes herramientas software empleadas tanto para su control como para la detección de los brazos y manos.

3.1 Cámara Kinect

3.1.1 Descripción

Azure Kinect es un kit de desarrollo de informática espacial de vanguardia con sofisticados modelos de voz y visión artificial, sensores de inteligencia artificial avanzados y una gran variedad de SDK con un gran potencial que se pueden conectar a Azure Cognitive Services [1].

En la Figura 3.1 se puede observar en detalle los diferentes componentes de esta cámara donde la numeración se corresponde a:

1. Sensor de profundidad de 1 MP con opciones de campo de visión ancho y estrecho.
2. Matriz de 7 micrófonos.
3. Cámara de vídeo RGB con 12 MP.
4. Acelerómetro y giroscopio (IMU).
5. Conexiones de sincronización externas.

En particular para este trabajo se hace uso de los sensores de profundidad así como de la cámara RGB. Mencionar que, aunque se explica con más detalle en el apartado 4.2.1, todo lo

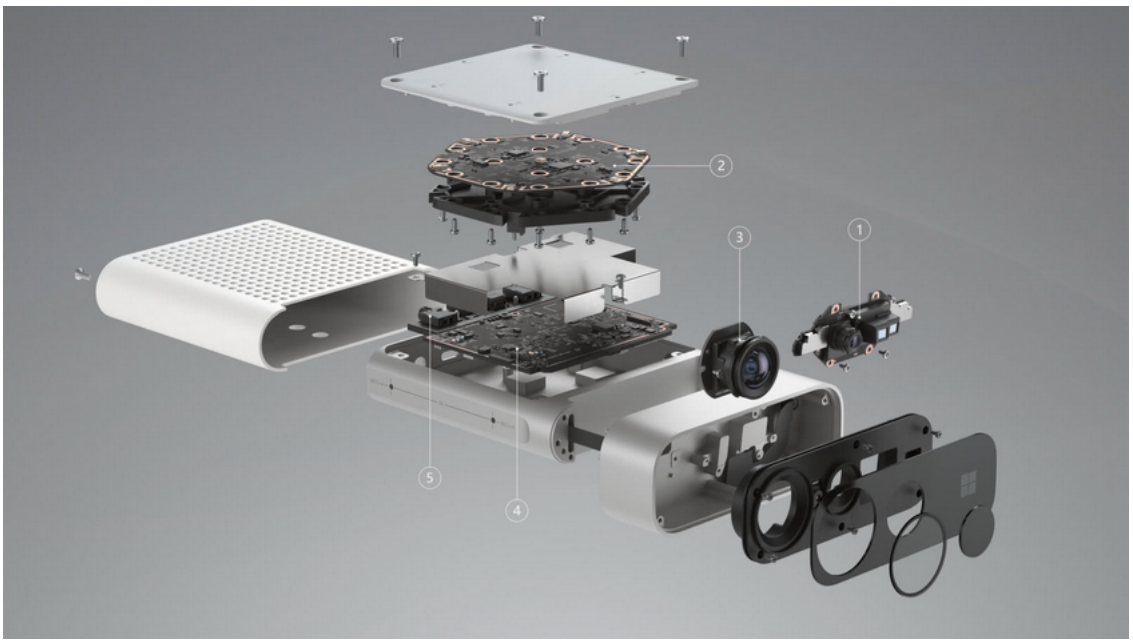


Figura 3.1: Contenido de Azure Kinect. Imagen obtenida de [1].

relativo a la programación de la Azure Kinect se realiza internamente empleando el lenguaje C/C++ pero por comodidad se ha optado por el uso de un *wrapper*[1] en Python.

3.1.2 Conexión y prueba de los sensores

El propio SDK proporcionado por Microsoft [21] incluye una herramienta (*K4aviewer*) la cual permite visualizar los diferentes sensores y comprobar su funcionamiento además de ajustar ciertos parámetros de las cámaras como la resolución, los Frames Por Segundo (FPS), el balance de blancos o el tiempo de exposición. Todas las pruebas se han realizado empleando Ubuntu 18.04 LTS como Sistema Operativo (SO) y la versión Kinect SDK 1.3.

En la Figura 3.2 se muestra un ejemplo de los diversos sensores, empleando el de profundidad con mayor campo de visión, pero menor distancia efectiva (sensor *Wide Field of View* (WFoV)). Por otro lado, en la Figura 3.3, se aprecia la misma configuración, pero en este caso el sensor de profundidad empleado es aquel que proporciona una mayor distancia efectiva, pero campo de visión más reducido (sensor *Narrow Field of View* (NFoV)). En las dos imágenes el sujeto se encuentra en la misma posición, con un brazo más adelantado que el otro, para poder apreciar bien las diferencias entre los sensores.

[1]Método para envolver código de un lenguaje en otro.

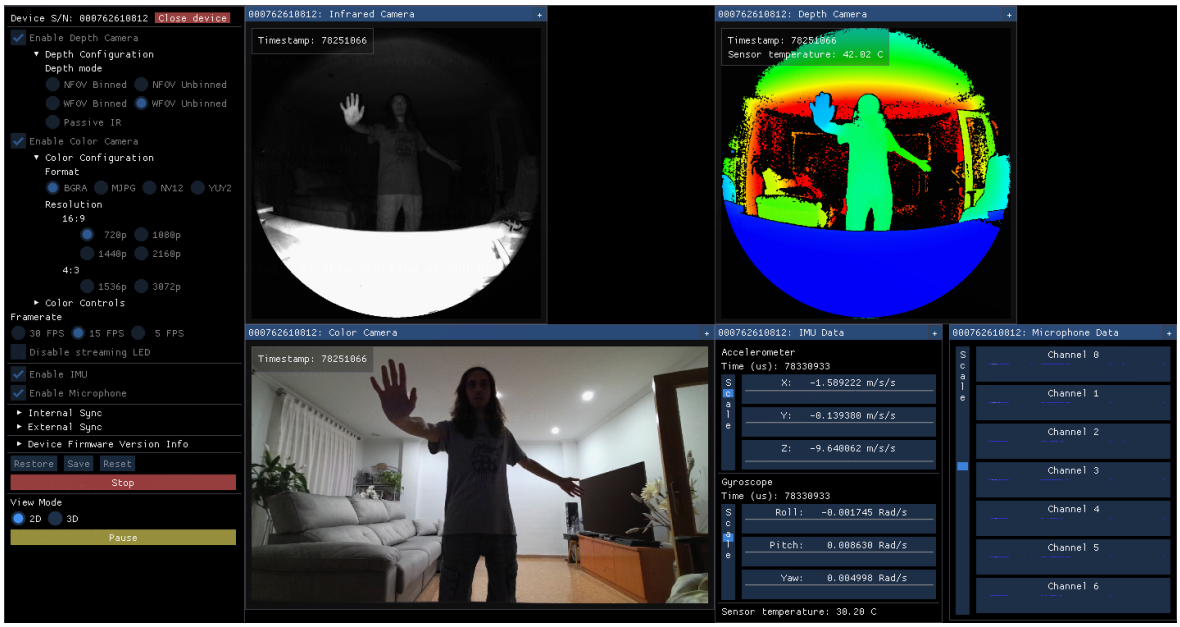


Figura 3.2: Sensor *Wide Field of View*.

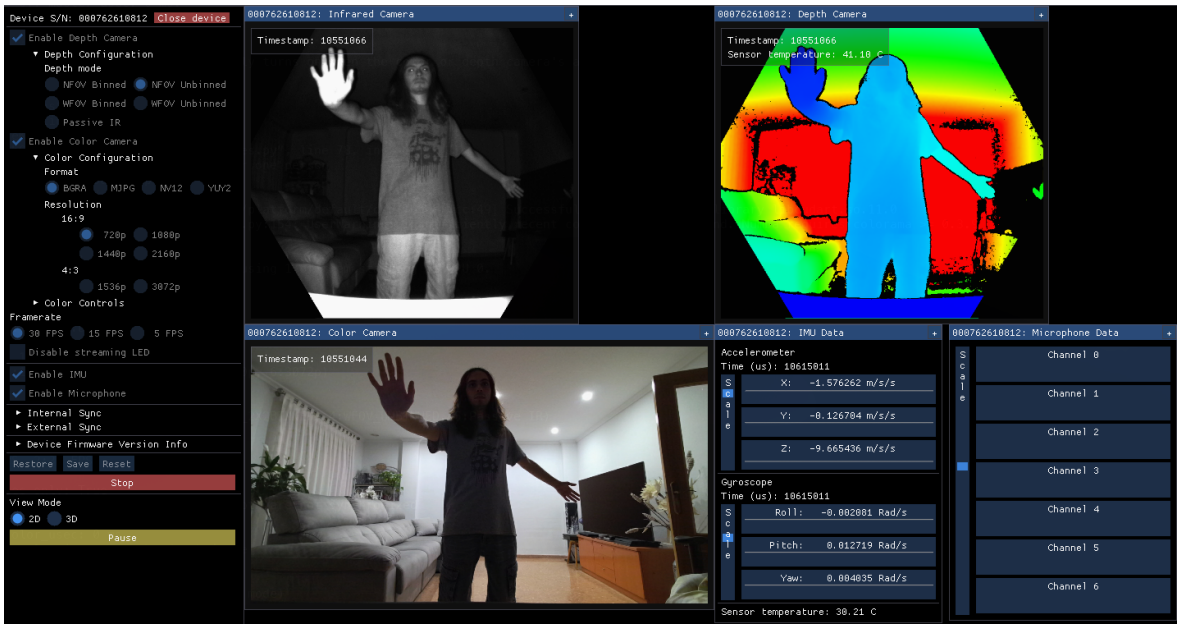


Figura 3.3: Sensor *Narrow Field of View*.

No es posible emplear ambos sensores a la vez, por lo que hay que decidir cuál de ellos es más acorde a la tarea a realizar. Además, dentro de cada tipo de sensor, se encuentran dos configuraciones diferentes, bien con la lente como *binned* o como *unbinned*.

En el modo *unbinned* se obtiene la imagen tal y como la captura el sensor, por otro lado en el modo *binned* se agrupan conjuntos de cuatro píxeles en uno, esto permite reducir el nivel de ruido obtenido en la imagen además de aumentar el brillo en condiciones de baja luminosidad pero reduciendo el tamaño total de la imagen en cuatro. La Tabla 3.1 resume las diferencias entre las distintas configuraciones.

Modo	Resolución	FoV	FPS	Rango	Tiempo exposición
NFOV unbinned	640x576	75°x65°	0, 5, 15, 30	0.5 - 3.86 m	12.8 ms
NFOV 2x2 binned	320x288	75°x65°	0, 5, 15, 30	0.5 - 5.46 m	12.8 ms
WFOV unbinned	1024x1024	120°x120°	0, 5, 15	0.25 - 2.21 m	20.3 ms
WFOV 2x2 binned	512x512	120°x120°	0, 5, 15, 30	0.25 - 2.88 m	12.8 ms

Tabla 3.1: Características sensores de profundidad. Obtenido de [7].

3.1.3 Body tracking

Junto a la capacidad de interactuar con los diferentes sensores mediante el Azure Kinect Sensor SDK [21], el kit de desarrollo de Microsoft incluye también un conjunto de librerías dedicadas al seguimiento del cuerpo (*Body Tracking*). Dichas librerías permiten la segmentación del cuerpo, la identificación de diferentes usuarios, así como la localización de las distintas articulaciones que forman el cuerpo humano.

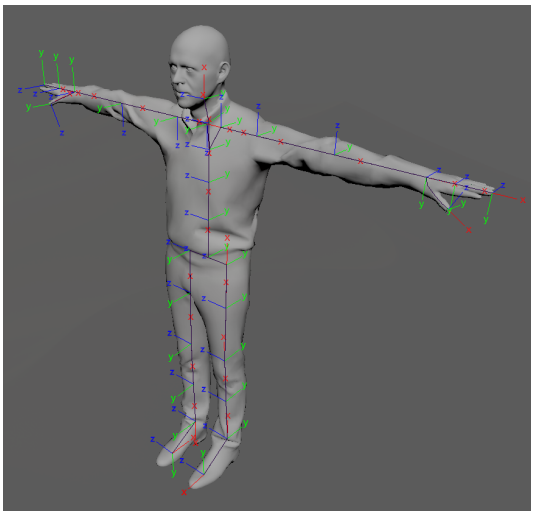
La segmentación del cuerpo se realiza empleando la imagen de profundidad o infrarroja donde se clasifica cada píxel como fondo o con el número identificativo de cada persona (en caso de haber varias) a la que posteriormente se le asigna un color para una mejor visualización (ver Figura 3.4).

Para la localización de las articulaciones se crea un sistema de coordenadas para cada articulación (ver Figura 3.5a) de manera que las coordenadas de todas ellas se estiman con respecto al sistema de referencia situado en el sensor de profundidad. De esta manera se genera una jerarquía de las diferentes articulaciones presentes (Figura 3.5b) donde las uniones (brazos) son conexiones entre el *joint* padre y su hijo. Todo este procesamiento se realiza de

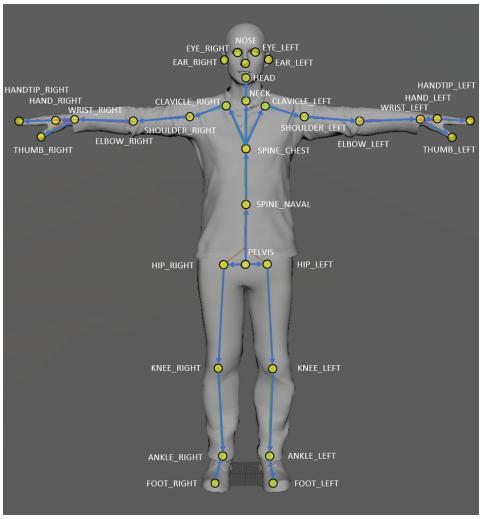
Figura 3.4: Segmentación del cuerpo. Imagen obtenida de [2]

manera interna mediante una RNA a la que se le introduce la imagen 2D obtenida desde la cámara infrarroja, calcula las coordenadas de los puntos de interés, y, posteriormente, a cada píxel 2D se le asigna el valor correspondiente obtenido del sensor de profundidad, obteniendo un punto 3D (x, y, z) como resultado. Finalmente, estos puntos se transforman a coordenadas del mundo situando el origen de coordenadas en el centro del sensor de profundidad, como se puede ver en la Figura 3.6.

A lo largo de este proyecto se ha estudiado la viabilidad de emplear este sistema de seguimiento por su elevada velocidad y precisión aunque sólo fuese para la parte de los brazos ya que, como se puede ver en la Figura 3.7, el sistema no detecta los dedos de manera individual (cuestión indispensable en este caso). La idea ha sido rechazada ya que al ser necesarios los dedos, el sistema debía pasar por otra herramienta como Openpose (sección 3.2) la cual requiere también de todo el esqueleto para obtener estos puntos. Por tanto simplemente se emplea cada uno de los sensores de manera independiente.



(a) Sistemas de coordenadas de cada articulación. Imagen obtenida de [22].



(b) Jerarquía de las articulaciones. Imagen obtenida de [3]

Figura 3.5: Modelado de puntos *Kinect Body Tracking*.

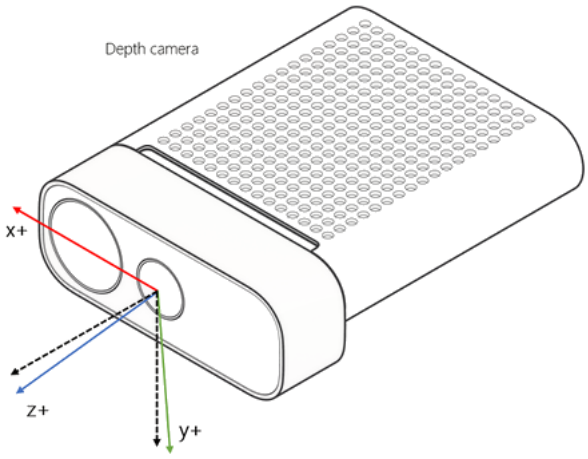


Figura 3.6: Origen del sistema de referencia. Imagen obtenida de [3]

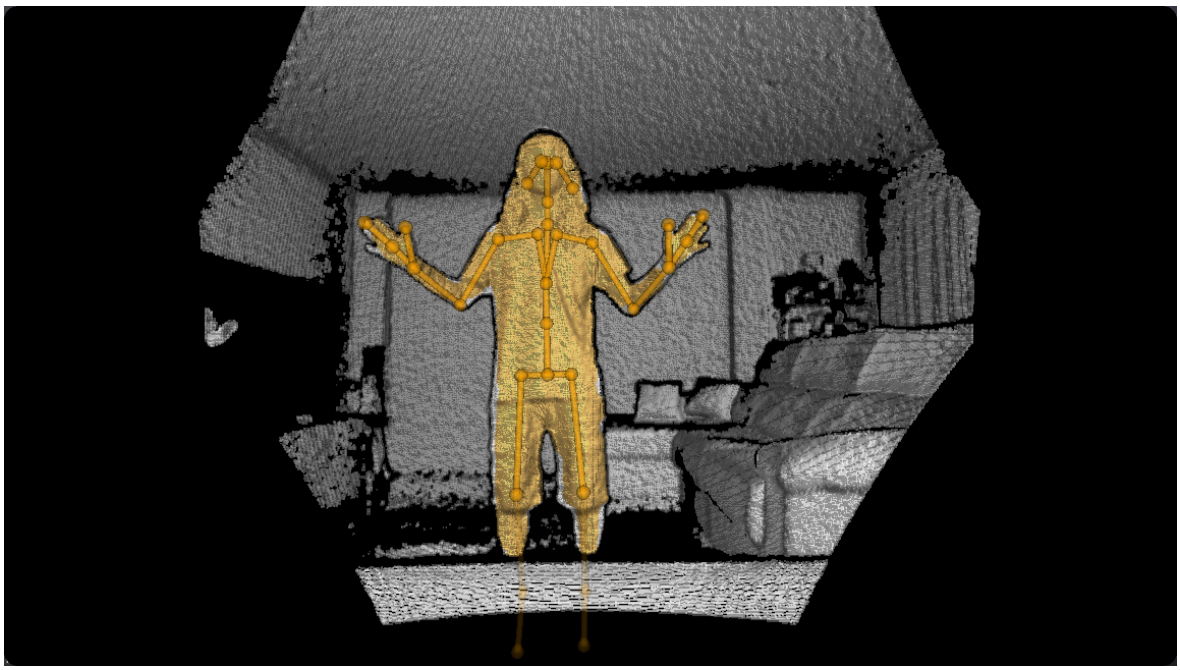
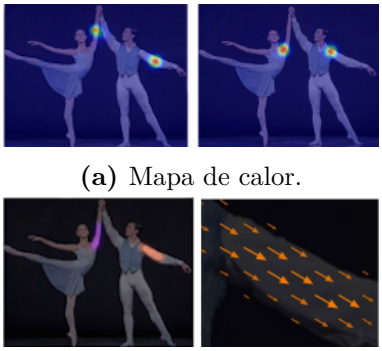


Figura 3.7: *Kinect Body Tracking.*

3.2 Openpose

Openpose es un sistema de detección de cuerpo, cara, manos y pies en tiempo real mediante diferentes puntos de interés (*keypoints*), obteniendo un total de 135 en el caso de la detección completa.

Este proyecto sólo se ha centrado en la detección de los *keypoints* relativos a los brazos y manos haciendo uso de las diferentes herramientas facilitadas por Openpose [23]. Dichas herramientas proporcionan la capacidad de capturar los diferentes *keypoints* en base a mapas de calor y de confianza como se puede apreciar en la Figura 3.8. De esta manera se calculan los mapas de calor (3.8a) y un set de vectores 2D de los mapas de afinidad (Figura 3.8b) que codifica el grado de asociación entre las partes (Figura 3.8c) [4]. Adicionalmente, en la Figura 3.9 se puede observar el número total de puntos de interés que es capaz de obtener Openpose a partir de una sola imagen 2D.

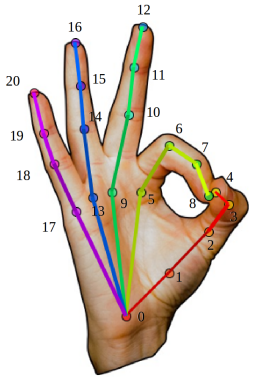


(a) Mapa de calor.

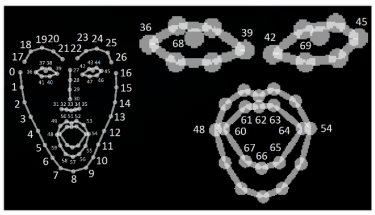
(b) Mapa de afinidad.

(c) Unión de puntos.

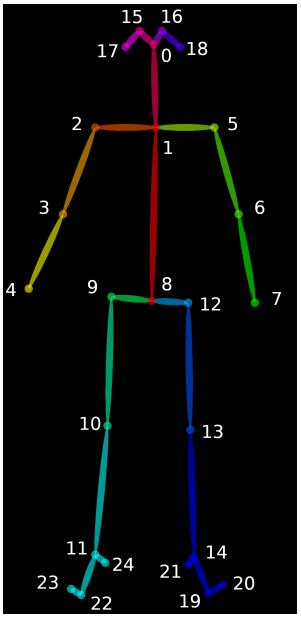
Figura 3.8: Mapas de calor. Imágenes obtenidas de [4].



(a) Mano.



(b) Cara.

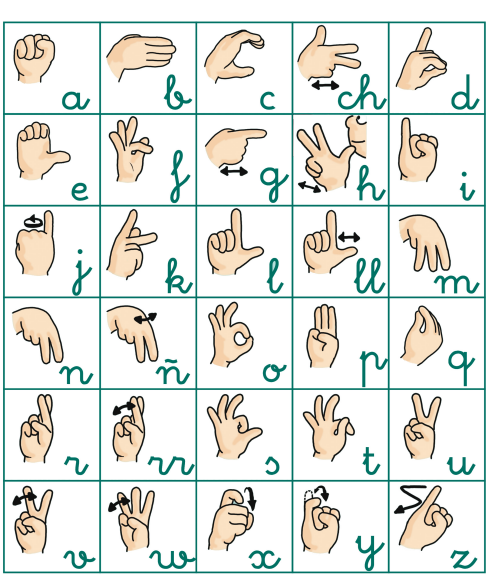


(c) Cuerpo.

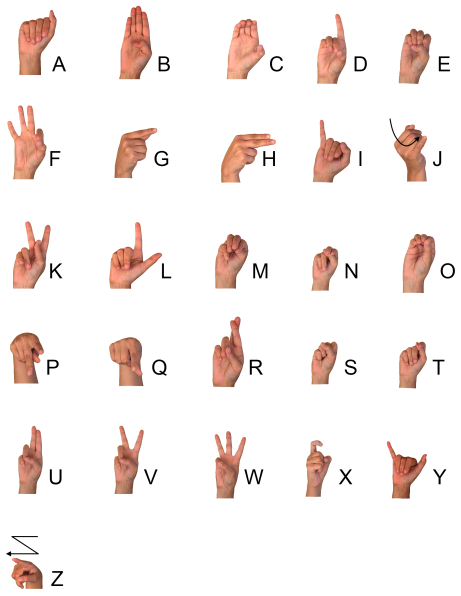
Figura 3.9: Total de *keypoints*. Imágenes obtenidas de [4] y [5].

4 Obtención del conjunto de datos

Uno de los principales inconvenientes que se han encontrado en la ejecución de este proyecto es la carencia de bases de imágenes (*dataset*) para la lengua de signos castellana, lo cual ha obligado a la creación de ésta. La falta de un *dataset* generalizado y los problemas de emplear uno de otro país se debe a las grandes diferencias entre la lengua de signos de un país a otro e incluso dentro de las diferentes comunidades de un mismo país. Un ejemplo de esta diferencia se puede observar en la Figura 4.1 donde se muestran los alfabetos dactilológicos castellano y americano.



(a) Castellano.



(b) Americano.

Figura 4.1: Diferencia entre alfabetos en lengua de signos en función del idioma.

Como se ha mencionado al comienzo de esta memoria, el objetivo es interpretar palabras y/o frases relativas a actividades diarias y, en particular, nos hemos centrado en el vocabulario

relativo a la alimentación. Así, la captura de datos se ha estructurado en diferentes categorías, cada una de las cuales está compuesta por distintas palabras. El conjunto total de palabras que se han considerado en este proyecto son:

- **Pronombres:** yo.
- **Nombres:** carne, cuchara, cuchillo, hamburguesa, huevos, pescado, pizza, plato, sopa, tenedor, tortilla, vaso, verdura.
- **Verbos:** cenar, comer, desayunar, gustar, no gustar, querer.
- **Partículas interrogativas:** cuándo, dónde, qué.

Todo esto permite múltiples combinaciones de frases cotidianas tales como *"¿Dónde está el tenedor?"*, *"¿Cuándo comemos?"*, *"De comer quiero huevos"* o *"No me gusta la tortilla"*.

Para la creación del *dataset* se ha empleado la cámara Azure Kinect y la herramienta OpenPose tal y como se ha comentado en el capítulo 3. En la Figura 4.2 se muestra el diagrama de flujo seguido a lo largo del proceso de creación del dataset correspondiente a este conjunto de datos.

4.1 Introducción LSE

Dentro de la Lengua de Signos Española las frases no se forman de igual manera que se hace en el castellano hablado o escrito, si no que se sigue una estructura diferente.

En la LSE las frases se construyen siguiendo la estructura *sujeto + acción + verbo + partícula interrogativa* en contraposición a la lengua castellana donde se emplea *partícula interrogativa + sujeto + verbo + acción*. Además hay una serie de verbos que se omiten siempre como son *ser*, *estar* o *tener* mientras que el sujeto, por norma general, siempre se indica. Caso especial reciben las negaciones de los verbos que en algunos casos (como *no gustar*) es una modificación del propio signo y, por tanto, se hace al signar el verbo, mientras que en otros casos la negación se añade al final de la frase.

Con esta breve introducción a la estructura de las frases se pretende dar sentido a la construcción del *dataset*, ya que para la creación de sentencias simplemente es necesario "unir"

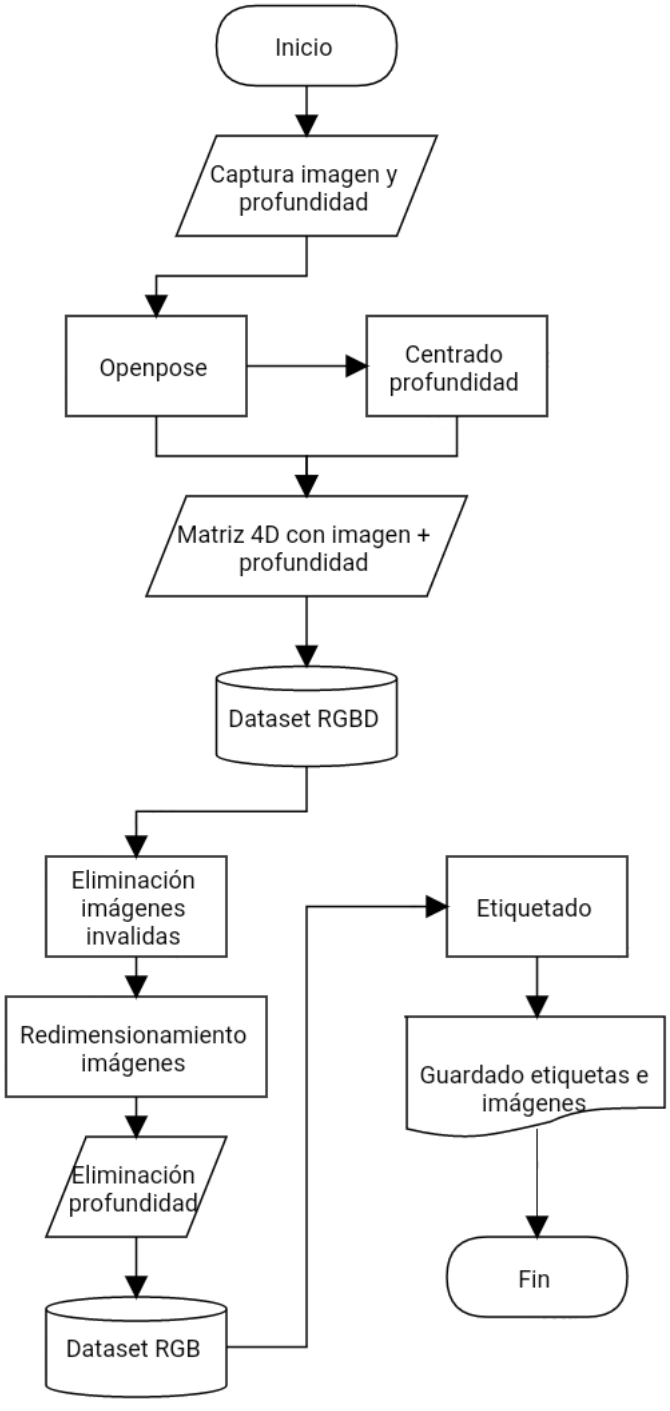


Figura 4.2: Diagrama de flujo de la creación del *dataset*.

cierto conjunto de signos. Un ejemplo de algunas oraciones en castellano y su correspondiente traducción en LSE se puede encontrar en la Tabla 4.1.

Frase castellano	**Frase en LSE**
¿Dónde está el tenedor?	Tenedor + dónde
¿Cuándo como?	Yo + comer + cuándo
De cenar quiero huevos	Yo + huevos + cenar + quiero
No me gusta la tortilla	Yo + tortilla + no-gustar

Tabla 4.1: Ejemplo de frases en castellano y en LSE.

4.2 Configuración de la cámara

El primer paso para la creación del *dataset* es la correcta comunicación y configuración de la cámara Azure Kinect así como el control de la iluminación en el entorno de trabajo.

Respecto a la posición la cámara, ésta se encuentra situada a 87 cm respecto del suelo (medido desde el centro del sensor de profundidad) y con una elevación de 10.1 ° respecto a la horizontal. Los sujetos se sitúan a una distancia entre 1.6 y 2.2 m dependiendo de la altura del individuo. En la Figura 4.3 puede observarse un ejemplo de lo que observan las cámaras tanto de color como de profundidad siguiendo esta configuración.

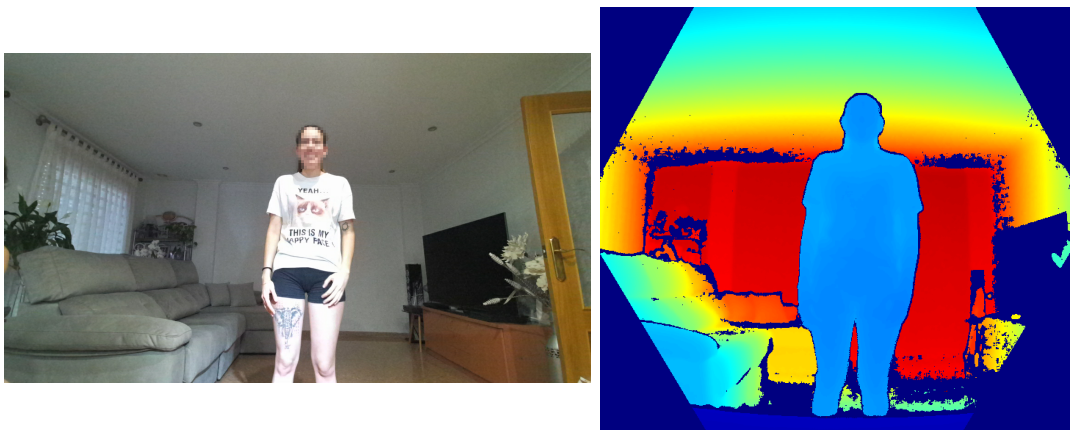


Figura 4.3: Ejemplo cámaras de color y profundidad.

Además de la posición de la cámara, es necesario controlar la iluminación con el fin de mejorar la extracción de las características visuales para el reconocimiento del signado. A

este respecto, se realizan algunos ajustes en el balance de blancos y el tiempo de exposición mediante la herramienta *k4aviewer* que ayudan a corregir los problemas derivados de procesar las imágenes con Openpose (más detalle en la sección 4.5). La Figura 4.4 muestra la configuración de las diferentes lentes utilizada a lo largo del proyecto, así como la calidad de la imagen o el número de FPS empleados.

```
Device configuration:
        color_format: 3
        (0:JPG, 1:NV12, 2:YUY2, 3:BGRA32)

        color_resolution: 1
        (0:OFF, 1:720p, 2:1080p, 3:1440p, 4:1536p, 5:2160p, 6:3072p)

        depth_mode: 2
        (0:OFF, 1:NFOV_2X2BINNED, 2:NFOV_UNBINNED,3:WFOV_2X2BINNED, 4:WFOV_UNBINNED, 5:Passive IR)

        camera_fps: 2
        (0:5 FPS, 1:15 FPS, 2:30 FPS)

        synchronized_images_only: True
        (True of False). Drop images if the color and depth are not synchronized

        depth_delay_off_color_usec: 0 ms.
        Delay between the color image and the depth image

        wired_sync_mode: 0
        (0:Standalone mode, 1:Master mode, 2:Subordinate mode)

        subordinate_delay_off_master_usec: 0 ms.
        The external synchronization timing.

        disable_streaming_indicator: False
        (True or False). Streaming indicator automatically turns on when the color or depth camera's are in use.
```

Figura 4.4: Parámetros de configuración empleados.

4.2.1 Programación de la cámara

La librería Azure Kinect SDK emplea el lenguaje C/C++ para trabajar con cada uno de los sensores, realizar transformaciones entre las imágenes de unos sensores a otros, calibrar las cámaras, etc. pero por comodidad a la hora de integrarlo todo posteriormente con Openpose y Tensorflow se ha optado por emplear un *wrapper* de Python.

La idea detrás del *wrapper* de Python es poder controlar las diferentes funciones creadas para la Azure Kinect desde este otro lenguaje. Se puede entender como un "traductor" de un lenguaje a otro (Python a C/C++).

Durante la búsqueda de un *wrapper* desarrollado (parcial o totalmente) se han encontrado únicamente tres modelos. Por un lado, [24] es una librería sencilla de usar que encapsula el código de C++ mediante librerías existentes como `Python.h` lo que permite que, desde Python, las imágenes devueltas por los sensores sean tratados como matrices de Numpy y objetos de Python. Por otro lado, [25] no redefine funciones para integrarlas en Python si no

que hace uso de la librería `ctypes`[1], de esta manera todos los datos y funciones empleadas son las originales de C usando Python únicamente como paso intermedio. Finalmente, [26] consiste en una combinación de los dos anteriores, permitiendo tratar los datos de la manera original en C o como objetos de Python.

Por la versatilidad de este último, además de presentar una mayor documentación que los otros es el *wrapper* empleado a lo largo del proyecto. Hay que mencionar que, aunque no se comenten en profundidad los cambios realizados, la versión original de Github se ha modificado para ajustarse a las necesidades de este proyecto. Así, se han implementado ciertos métodos para obtener datos como la calibración de las cámaras (tiempos de exposición, balance de blancos, tamaño de la imagen, etc.) y redefinido algunos existentes para la generación de transformaciones de unas imágenes a otras así como para adaptarlo a la versión del Kinect SDK empleado.

4.2.1.1 Obtención de imágenes

Una vez solventado el problema de la diferencia de lenguajes de programación, se procede a la captura de imágenes. No se van a detallar las funciones utilizadas ya que se encuentra todo documentado en la propia API de Microsoft [21] así como en el *wrapper* empleado [26] pero sí se va a explicar el flujo seguido. Las funciones nombradas a lo largo de esta sección hacen referencia a las implementadas en la API [21] a no ser que se indique lo contrario.

Inicialmente se configuran los parámetros de las lentes. En este caso se emplea el sensor de profundidad NFoV en modo *unbinned* y la cámara de color con formato BGRA de 32 bits, una resolución de 720p y 30 FPS (Figura 4.4). Además, ambas imágenes han de estar sincronizadas, lo que implica que, de manera interna, se desechan aquellas imágenes de una u otra lente hasta que ambas se capturan en el mismo instante de tiempo.

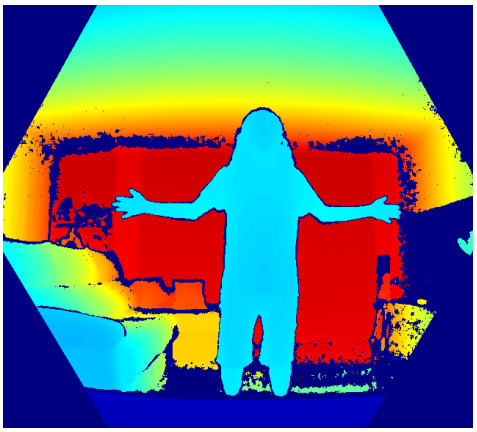
Con estos parámetros se inician las cámaras (función `k4a_device_start_cameras`) y se calcula el valor de calibración entre el sensor de profundidad y la cámara de color empleando la función `k4a_device_get_calibration`. Este valor se emplea para realizar transformaciones de un sistema de referencia al otro (cada sensor tiene el suyo propio ubicado en su centro geométrico). Este proceso es de gran importancia ya que las imágenes no son del mismo

[1]Proporciona tipos de datos compatibles con C y permite llamadas a funciones en librerías compartidas.

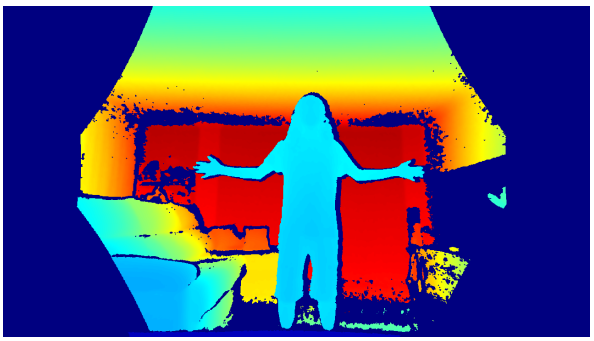
tamaño y, por tanto, hay que convertirlas o bien al tamaño de la imagen de color (más grande) o a la de profundidad (más pequeño). La Figura 4.5 muestra la imagen original en color, la original en profundidad y la convertida de profundidad a color.

Finalmente, se comienzan a capturar las imágenes (`k4a_capture_get_color_image` y `k4a_capture_get_gepth_image`). Éstas se pasan a la función `k4a_transformation_depth_image_to_color_camera` que se encarga de generar las transformaciones mencionadas de una cámara a otra, que en este caso es de la imagen de profundidad a la de color. Por último, las dos imágenes obtenidas, la imagen en color y la de profundidad transformada, se convierten en matrices de Numpy (función `image_convert_to_numpy` del *wrapper* [26]) para poder trabajar con ellas. En el caso de la matriz de profundidad obtenida las distancias están medidas en milímetros respecto de la cámara.

(a) Imagen capturada por la cámara de color.



(b) Imagen capturada por por el sensor de profundidad.



(c) Imagen transformada del sensor de profundidad a la cámara de color.

Figura 4.5: Ejemplo de imágenes capturadas por los sensores.

4.3 Análisis de la imagen

Una vez obtenida la imagen ésta se procesa con Openpose para extraer todos los puntos característicos necesarios. En este momento solo se está trabajando con la imagen en color. Aunque la herramienta Openpose permite el almacenamiento de cada uno de estos *keypoints* en ficheros con extensión *json*, en este caso se ha trabajado sobre los puntos que va generando en tiempo real.

4.3.1 Mano

Con todos los puntos de interés obtenidos, se procede a analizar la posición de la mano a la que corresponde cada punto. Openpose genera 21 puntos para cada mano, de los cuales cada punto es un vector de tres valores que se corresponden con las coordenadas X e Y y un valor de puntuación que indica la certeza que se tiene de que ese punto esté en la posición indicada. De esta manera, se genera un vector de 63 valores para cada mano.

En la Figura 4.6 se puede observar el orden y posición de los diferentes puntos extraídos por Openpose. Conociendo esto y que cada punto en la imagen se corresponde con tres valores en el vector generado (X, Y, confianza), se puede dibujar en la imagen que se está analizando las articulaciones y las falanges.

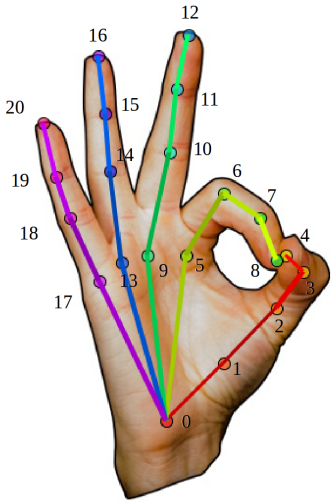
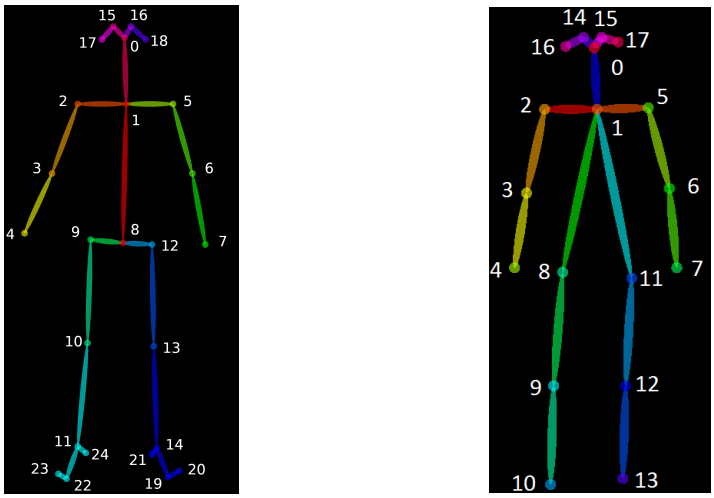


Figura 4.6: Puntos Openpose.

4.3.2 Brazo

De manera similar a como se ha realizado el análisis de la mano, se analizan los *keypoints* de los brazos. En este caso Openpose tiene dos maneras diferentes de calcular los puntos, dependiendo del modelo que se emplee para el cuerpo. Los dos modelos más comunes son COCO y BODY_25 (Figura 4.7), aunque para este proyecto en particular es indiferente el modelo que se use, ya que los puntos que se buscan no difieren de un modelo a otro. No obstante, se ha empleado el modelo BODY_25, ya que produce mejores resultados si se trabaja sobre una GPU, mientras que en el caso de trabajar sobre CPU el modelo COCO es mejor [27].

Igual que se ha hecho en el caso de las manos, se dibujan las articulaciones y los huesos sobre la imagen, tal y como se puede ver en la Figura 4.9c.



(a) BODY_25. **(b)** COCO.

Figura 4.7: Modelos de cuerpo.

4.3.3 Unión brazo-mano

Una vez se han calculado y comprobado correctamente que los puntos dibujados se corresponden con las articulaciones deseadas, se pueden dibujar todos los puntos juntos. En este caso los puntos se introducen en una matriz con cuatro dimensiones (RGB-D) donde las tres primeras se corresponden con los canales de color y la cuarta se reserva para la profundidad.

Además, para evitar problemas con los valores de profundidad esta matriz tiene que ser de tipo *float*.

Respecto a la parte de color, se emplea un fondo negro y cada lado se codifica con un color distinto, rojo para el lado derecho y azul para el izquierdo como se puede apreciar en la Figura 4.9d.

4.3.4 Profundidad

Dado que la matriz de profundidad obtenida tiene su referencia en el centro de la cámara, los valores son dependientes de la distancia de la persona con respecto a la cámara en lugar de la distancia entre manos y brazos a la hora de signar. Por tanto, todos estos valores se cambian de sistema de referencia a uno situado sobre el sujeto.

Para ello, empleando Openpose, se extrae la posición X e Y del cuello y la cadera y se calcula la distancia media entre éstos. Esto devuelve el punto medio del cuerpo que será el nuevo sistema de referencia (Figura 4.8). A continuación, se calcula la distancia desde el sensor hasta ese nuevo punto y se considera como la nueva distancia origen, añadiéndole un pequeño *offset* como margen de error. Además, se multiplica por -1 para considerar positivos los movimientos del pecho hacia delante y negativos al contrario. Esto último es simplemente para poder visualizar los datos de manera más intuitiva.

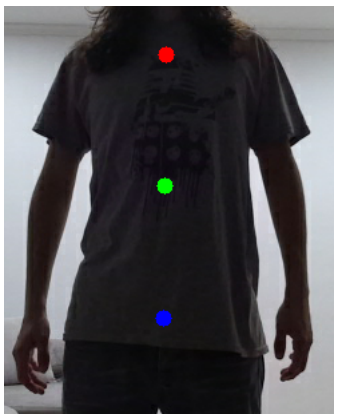
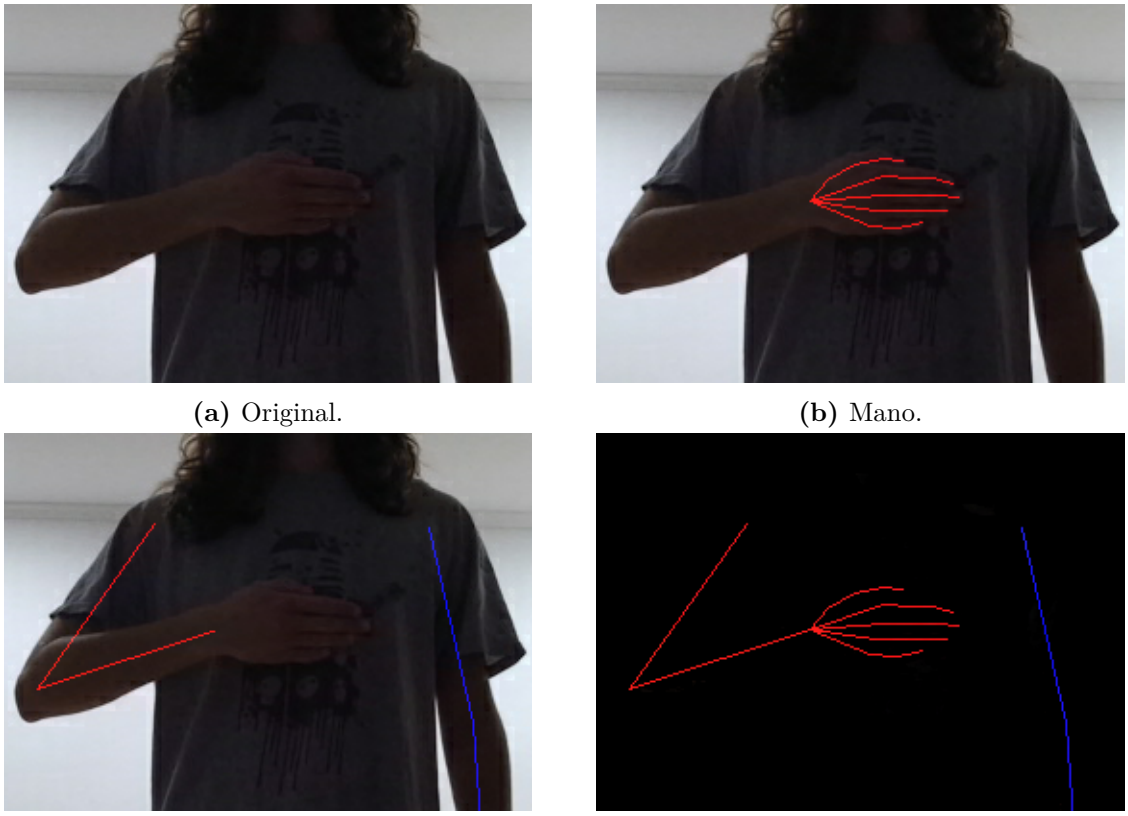


Figura 4.8: En rojo el cuello, azul la cadera y verde el centro entre ellos.

De esta manera cada movimiento que se realice es independiente de la distancia a la que se encuentra la persona de la cámara, solo se tiene en cuenta la distancia del cuerpo a las

manos, la cual tiende a variar poco entre unos sujetos y otros.

Finalmente todos estos valores son introducidos en la última dimensión de la matriz previamente creada, cada uno en el punto (X, Y) que corresponda. Obteniendo así una matriz con los brazos y manos codificados en colores y la distancia de cada uno de estos píxeles respecto del cuerpo. Dicha matriz es la que se almacena en formato Numpy (extensión *npy*).



(a) Original. **(b)** Mano.

(c) Brazo. **(d)** Completo.

Figura 4.9: Proceso de análisis de la imagen.

4.4 Clasificación de las imágenes

Comprobado que el análisis de las imágenes funciona de manera correcta, se procede a clasificarlas. Para ello se han realizado dos grabaciones, por un lado los signos realizados con la mano derecha y por otro los realizados con la mano izquierda. Se ha hecho de esta manera debido a que dependiendo de si una persona es diestra o zurda realizará el gesto con

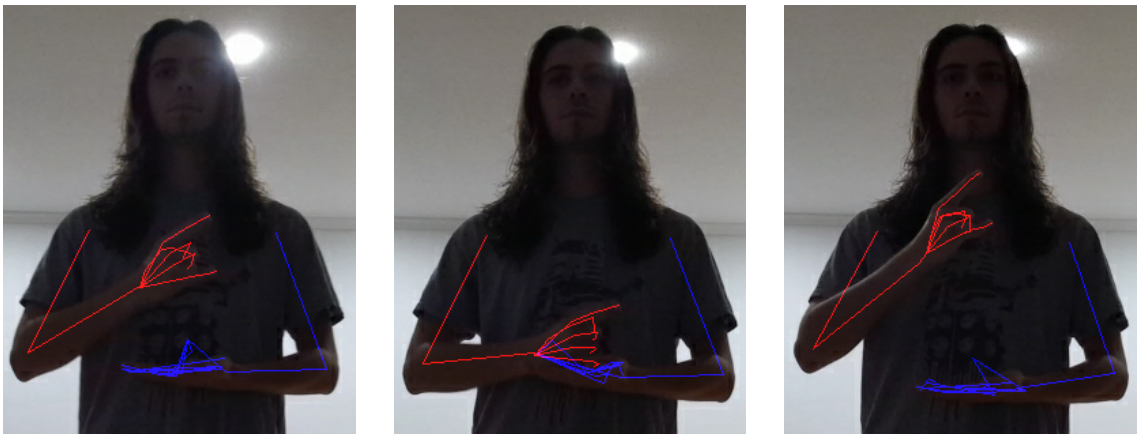
la mano dominante. Esto no se ha realizado con todos los signos ya que hay algunos que son bimanuales, es decir, que en el signado se emplean ambas manos por igual, en estos casos se ha duplicado el tiempo de grabación.

En la creación del *dataset* han participado un total de 5 personas (2 mujeres y 3 hombres) las cuales han sido grabadas durante un intervalo de 60-80 segundos por signo y por mano con excepción de algunas secuencias más largas como es el caso del verbo *no gustar* el cual ha requerido de grabaciones de 120-140 segundos.

El tiempo aproximado invertido en todo el proceso de calcular los *keypoints*, normalizar las distancias, almacenarlo en la nueva matriz y guardarlo en disco se sitúa en un intervalo entre 0.21-0.26 segundos, por tanto la tasa de guardado de datos es de unas cuatro matrices por segundo.

Posteriormente, dichas imágenes deben de analizarse y separarse en secuencias de signos de longitud variable según la persona, la mano y el signo. Además, hay signos que es necesario realizar dos veces mientras que otros se pueden realizar una, dos o tres veces sin cambiar su significado lo cual genera un ligero desbalanceo en los datos recogidos. Esto se debe a que en esta fase inicial de clasificación para aquellos signos que pueden realizarse más de una vez la secuencia recogida es de uno en uno.

En la Tabla 4.2 se puede observar el número total de imágenes guardadas para cada signo y su clasificación en secuencias. Mencionar que no es el número de imágenes grabado, si no que es el guardado una vez se ha realizado la separación y eliminación de aquellas imágenes erróneas. Además, en las Figuras 4.10, 4.11 y 4.12 se pueden ver algunos ejemplos de las secuencias grabadas para diferentes signos y personas.



(a) Inicio del signo. **(b)** Punto medio del signo. **(c)** Final del signo.

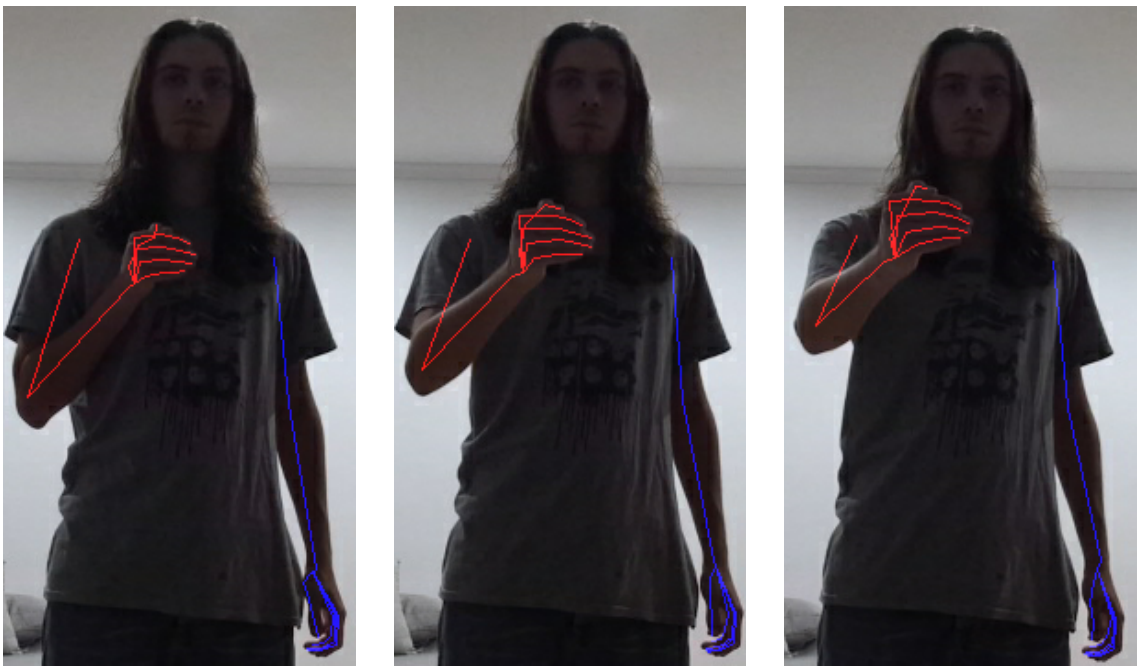
Figura 4.10: Signo para la palabra carne.

(a) Inicio del signo. **(b)** Punto medio del signo. **(c)** Final del signo.

Figura 4.11: Signo para la palabra huevos.

Signo	Número imágenes	Número secuencias
carne	1813	158
cenar	1685	129
comer	1722	142
cuando	1354	135
cuchara	1580	117
cuchillo	1483	192
desayunar	920	84
donde	1854	213
gustar	1633	95
hamburguesa	1214	145
huevos	925	96
no gustar	2545	105
pescado	1862	91
pizza	1481	50
plato	769	75
que	1402	120
querer	1545	113
sopa	1453	143
tenedor	1584	108
tortilla	2040	290
vaso	2071	96
verdura	1965	75
yo	1524	114

Tabla 4.2: Número de imágenes y secuencias por signo.



(a) Inicio del signo. **(b)** Punto medio del signo. **(c)** Final del signo.

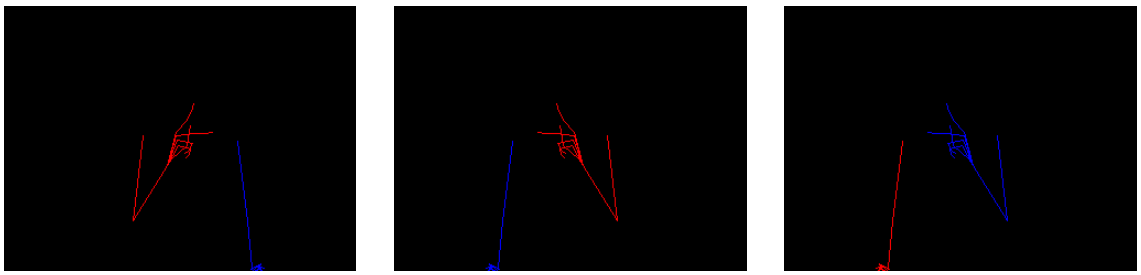
Figura 4.12: Signo para la palabra cuando.

4.4.1 Aumento del *dataset*

Adicionalmente se ha barajado la opción de aumentar el conjunto de datos generando imágenes espejo de cada una de las imágenes obtenidas. Esto es posible ya que, como se ha comentado al comienzo, los signos se graban con ambas manos por tanto se pueden generar nuevos datos para una mano empleando los de la otra.

En la Figura 4.13 se puede ver el proceso seguido. Primero se voltea verticalmente la imagen (Figura 4.13b) y después se cambian los colores de los brazos (Figura 4.13c) para que lo que antes era una imagen generada con la mano derecha ahora se corresponda con una imagen de la mano izquierda.

Aunque en la fase inicial de creación del *dataset* no se ha llegado a realizar la ampliación sí se ha generado el código necesario para esto por si en los entrenamientos de los diferentes modelos fuese necesario disponer de un número mayor de datos.



(a) Imagen original. **(b)** Imagen original volteada. **(c)** Imagen final.

Figura 4.13: Proceso de generación de imágenes espejo.

4.5 Problemas

La captura del *dataset* no está exenta de problemas y el hecho de trabajar con un sensor de profundidad y uno de color aumenta el número de éstos. Por ello se puede subdividir esta sección en aquellos problemas ligados al *software* de reconocimiento (Openpose) y aquellos directamente relacionado con los sensores o la Azure Kinect como herramienta *hardware*.

4.5.1 Openpose

En este caso se trata de aquellos problemas que, aunque derivados de algo físico como la luz, se relacionan de manera directa con Openpose por las propias limitaciones del sistema en ciertos casos. Entre los ejemplos más repetidos se encuentran:

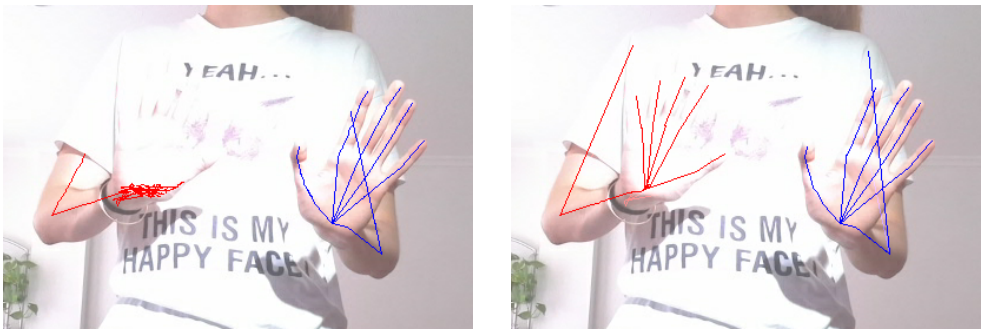
- **Iluminación**. Una fuente de luz directa sobre la cámara o un reflejo lumínico grande en la ropa de la persona genera falsos *keypoints* y en algunos casos no es capaz de detectar ninguno. En la Figura 4.14 se puede ver como, debido al color blanco de la ropa, se genera una luminosidad elevada y los *keypoints* obtenidos no son los deseados. Entre las soluciones tomadas se encuentran el control de la iluminación y el uso de ropa negra o gris para facilitar la captura.

- **Ángulo de los signos**. Ciertos signos es mejor grabarlos con un cierto ángulo a como se haría normalmente. Esto se hace para disminuir el número de oclusiones en la captura de la imagen y, por tanto, conseguir una mejor captura. Aunque en esta ocasión no se ha sido necesario modificar el ángulo ya que con la propia inclinación de la cámara es suficiente para corregir esas pequeñas oclusiones.

- **Relación mano-cara**. Uno de los problemas más recurrentes y que más datos ha obligado a desechar es la interacción entre la mano y la cara (necesaria en muchos de los signos). En muchos casos el acercar la mano al rostro hace que Openpose detecte falsos positivos y no sea capaz de "encontrar" la mano aunque previamente lo hubiese hecho. Un ejemplo de este error puede verse en la Figura 4.15a. Si bien es cierto que con una mayor corrección de la iluminación (Figura 4.15b) se consigue solventar ligeramente, es un problema que sigue apareciendo.

- **Relación mano-mano**. Similar al anterior, en algunas ocasiones al juntar ambas manos se producen oclusiones entre ellas lo que genera falsos *keypoints* (Figura 4.15c). Cabe destacar que en nuestro caso no se ha considerado un problema, por lo que no se han desechado los datos.

De los problemas mencionados se puede extraer que el sistema para el reconocimiento de manos no es el ideal para una situación real con iluminación variable, pero en un entorno controlado, como es el caso, es una opción factible aunque con limitaciones.

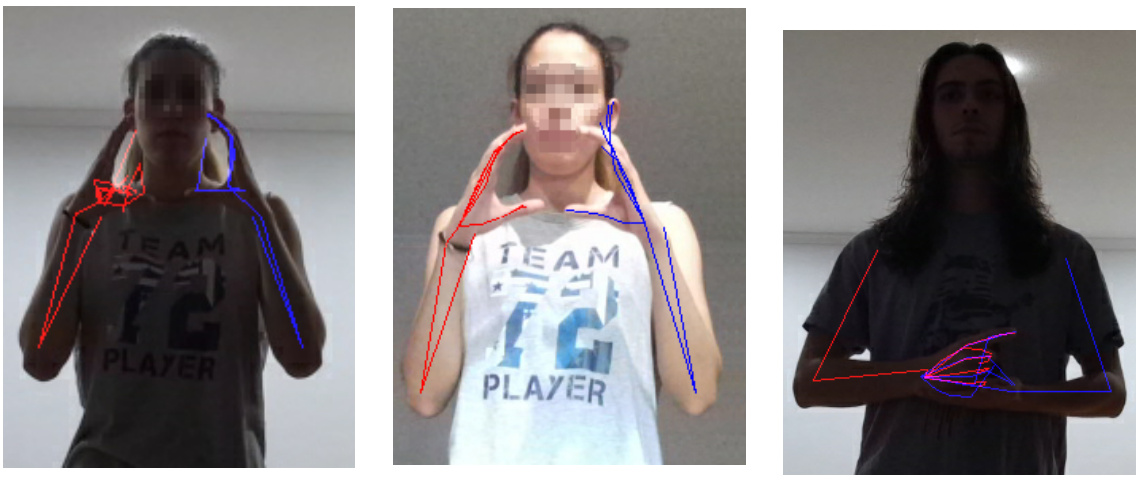


(a) *Keypoints* obtenidos. **(b)** *Keypoints* esperados.

Figura 4.14: Problemas debido a la iluminación.

4.5.2 Azure Kinect

Además de los inconvenientes iniciales normales como la colocación de la cámara o el ajuste de los parámetros, hay que destacar uno relacionado directamente con el sensor de profundidad (independiente de cual de ellos se utilice). Los sensores de profundidad miden dentro del espectro infrarrojo el tiempo que tarda la luz en recorrer la escena y volver (principio del



(a) Interacción mano-cara con gran error.

(b) Interacción mano-cara con menor error.

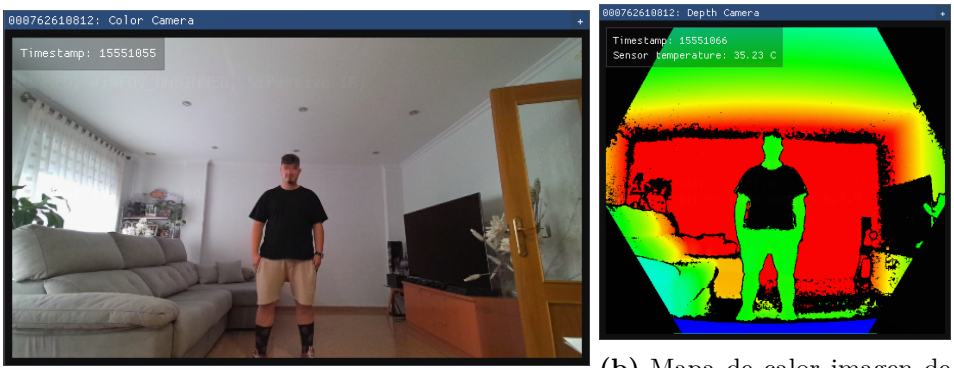
(c) Interacción mano-mano.

Figura 4.15: Problemas con la interacción entre las manos.

Time of Flight (ToF)) y de esta manera generan el mapa de profundidad.

Esto genera ciertos problemas ya que hay determinadas situaciones en las que no es capaz de detectar la distancia como, por ejemplo, cuando la señal infrarroja no tiene suficiente fuerza para iluminar la zona (mucha distancia) o cuando hay una señal infrarroja saturando. Estas situaciones en las que se anula se representan como zonas negras en el mapa de calor.

El único caso completamente invalidante que se ha encontrado es al usar ropa de poliéster donde es el sistema es incapaz de calcular la profundidad tal y como se puede observar en la Figura 4.16 debido a la alta absorbancia del material.



(a) Imagen en color.

(b) Mapa de calor imagen de profundidad.

Figura 4.16: Ejemplo de ropa no capturada por el sensor de profundidad.

El resto de pequeños problemas derivados del uso de la Kinect se encuentran más relacionados con el control de la API y el manejo a bajo nivel que con la parte física y no hay ninguno que requiera de mención especial.

5 Redes Neuronales Artificiales

A lo largo de este capítulo se plantean las diferentes arquitecturas estudiadas para la interpretación de los signos. Debido a que se trata de análisis de secuencias de datos se hace uso de Redes Neuronales Recurrentes ya que es necesario considerar la dimensión temporal además de la espacial.

5.1 Redes Neuronales Recurrentes

La peculiaridad de este tipo de redes es que poseen sus entradas y salidas conectadas en el tiempo. Es decir, la red recibe dos entradas, por un lado la del valor de la capa anterior y por otro, su propia salida en el instante de tiempo anterior. La Figura 5.1 muestra un ejemplo de esto donde el centro es una neurona, la x se corresponde con la entrada de la capa anterior y la y la salida producida.

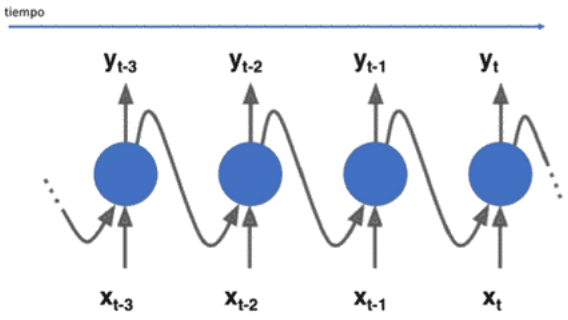


Figura 5.1: Ejemplo de neurona RNN desplegada. Imagen obtenida de [6].

El hecho de considerar como entrada el instante anterior permite a estas redes recordar lo visto en momentos anteriores. Esto ha hecho que sea un tipo de red comúnmente empleado en tareas como la traducción automática [28] o el reconocimiento de voz [29].

5.2 Long Short-Term Memory

Una de las arquitecturas más utilizadas son las conocidas como Long Short-Term Memory (LSTM) propuestas por Hochreiter y Schmidhuber [30] con la idea de recordar la información durante intervalos de tiempo mayores a la vez que trata de evitar el problema del desvanecimiento del gradiente, esto es, cuando el gradiente es propagado hacia las capas anteriores, el número de multiplicaciones que sufre hace que llegue a valores infinitesimales.

Para lograrlo utilizan las llamadas celdas de memoria (Figura 5.2) compuestas por tres puertas (*gates*): una para las entradas (*input gate*), otra para las salidas (*output gate*) y una última para olvidar (*forget gate*) que determina qué cantidad de información previa se considera en el estado actual.

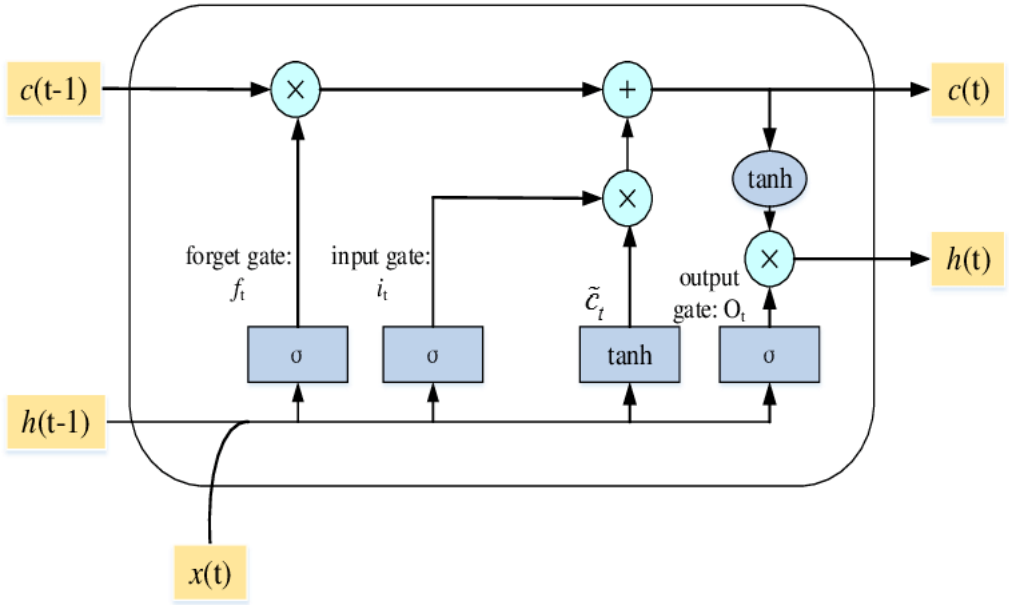


Figura 5.2: Celda de memoria de una LSTM.

En cada una de estas celdas, las variables $c(t-1)$ y $c(t)$ representan la memoria que se concatena de una celda a otra, las variables $h(t-1)$ y $h(t)$ muestran la salida de la celda anterior y de la actual, mientras que $x(t)$ indica la entrada en el instante actual. Respecto a las puertas, éstas pueden definirse como sigue:

- *Forget gate*: Entre la entrada actual y la salida previa calcula mediante una función Sigmoide la media de estas, devolviendo un valor en el rango [0, 1]. Este valor se

multiplica por la memoria del estado anterior para determinar la cantidad de esta que se mantiene a lo largo de la celda, olvidándola por completo si el valor devuelto es 0, manteniéndola intacta si es 1 o parcialmente en función del valor.

- *Input gate*: Recibe la entrada actual y la salida previa y aplican funciones de tipo Sigmoide y de Tanh (tangente hiperbólica). La activación Sigmoide es la encargada de aprender qué se debe mantener de la entrada mientras que la Tanh se encarga de normalizar el proceso en un rango de [-1, 1] añadiendo estabilidad al entrenamiento. Ambos valores son multiplicados antes de añadirlos a la memoria para asegurar esta normalización.

- *Output gate*: Multiplica el valor normalizado de la memoria actual y el valor entre la entrada y la salida previa con función Sigmoide determinando qué debe aprender para predecir la entrada actual. Este valor se devuelve como salida de la celda.

Al configurar las celdas de esta manera el desvanecimiento del gradiente se solventa ya que durante su cálculo (el cual sucede en el interior de la celda) solo se llevan a cabo operaciones lineales de adición y multiplicación.

Uno de los principales inconvenientes que plantean este tipo de capas es el elevado tiempo y memoria que requieren para su entrenamiento, además de sobreentrenarse con relativa facilidad. Por otro lado, tienden a presentar resultados muy precisos sobre todo en secuencias largas.

5.3 Gated Recurring Units

De manera similar a las LSTM, las capas Gated Recurring Units (GRU) [31] hacen uso del control del flujo de información mediante puertas dentro de celdas de memoria, pero de una manera más sencilla. Únicamente reciben y transmiten de la celda anterior la salida, en lugar de la salida y la memoria, esto hace que se trate de una arquitectura más simple y por ello más rápida. En la Figura 5.3 se muestra dicha celda de memoria.

La principal diferencia se encuentra en el número de puertas contenidas en la celda, dos en este caso. La puerta de reinicio (*reset gate*) y la de actualización (*update gate*). El funcionamiento de esta celda se puede definir como:

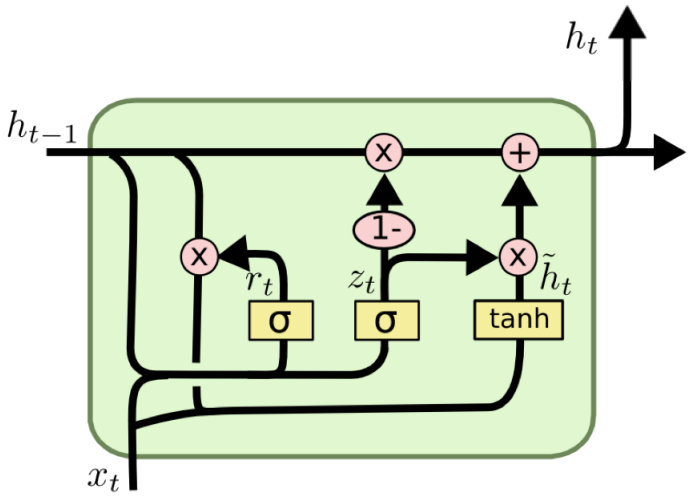


Figura 5.3: Celda de memoria de una GRU.

- *Reset gate*: Mediante una función Sigmoide se calcula aquello que se debe olvidar entre la entrada en el instante actual ($x(t)$) y la salida en el instante anterior ($h(t-1)$). Esto se añade directamente a $h(t-1)$ para excluirlo.

- *Update gate*: La entrada actual y la salida previa pasan por una función Tanh para normalizar su valor en el rango [-1, 1]. Este valor normalizado junto con el devuelto por la función Sigmoide entre la salida anterior y la entrada actual se añaden a $h(t-1)$. El valor resultante es la salida actual actualizada ($h(t)$).

Al tratarse de una arquitectura más sencilla que las LSTM su velocidad de entrenamiento y su consumo de memoria es menor, pero, en general, los valores de precisión obtenidos son inferiores, especialmente cuando se trata de secuencias temporales largas donde hay que almacenar mucha información en memoria.

5.4 Redes Neuronales Recurrentes bidireccionales

Un problema presente en las Redes Neuronales Recurrentes (RNNs) es que únicamente son capaces de recordar las relaciones entre los datos actuales y los pasados, pero en ocasiones es necesario comprender el contexto y, por tanto, aprender relaciones entre los datos presentes, pasados y futuros. En este punto entran las RNNs bidireccionales.

En este tipo de red se consideran dos secuencias separadas: por un lado, el flujo normal en el que la información se propaga hacia delante; y, por otro, la secuencia inversa en el que el flujo se propaga en sentido opuesto. Ejemplificando esto para la predicción de frases, si se asume que se tiene la secuencia *"Me gusta el café"*, la capa directa empleará la oración así mientras que la capa inversa utilizará *"café el gusta Me"*. De esta manera las salidas se actualizan considerando toda la dimensión temporal (presente, pasado y futuro). La Figura 5.4 muestra un ejemplo de la distribución de estas capas.

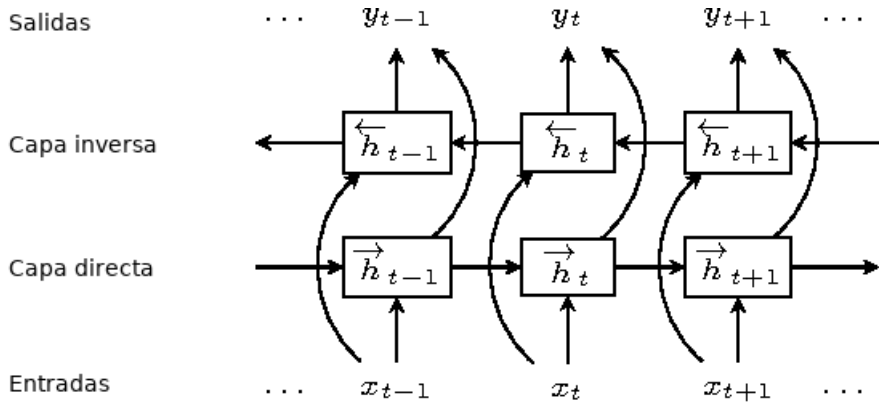


Figura 5.4: RNN bidireccional.

Estas redes bidireccionales permiten su uso junto con las capas LSTM y GRU vistas previamente aprovechando las ventajas de éstas como la memoria o precisión junto con la mejora en el entrenamiento al aprender un mayor rango temporal.

6 Experimentación

En este capítulo se van a comentar los resultados obtenidos con las diferentes arquitecturas expuestas en el capítulo 5, el método empleado para la carga de los datos así como los problemas surgidos.

Todos los experimentos se han realizado en un sistema con un procesador AMD Ryzen 7 5800X (8 núcleos), 32GB de RAM y una GPU NVIDIA GeForce GTX 1070 con 8GB de memoria. Para el entrenamiento de los modelos se ha empleado Tensorflow 2.4.0 sobre Python 3.6.

6.1 Carga de datos

Tanto las capas LSTM como GRU requieren de una entrada con tres dimensiones con la forma `[batch, timesteps, feature]` donde *batch* se corresponde con el número de secuencias, *tiemsteps* con la longitud de cada secuencia y *feature* es el conjunto de características de éstas.

Al trabajar con imágenes RGB las características (*feature*) se encuentran en una matriz con forma `[ancho, alto, canales]` por tanto se concatenan todas en un único vector para poder introducirlo en las redes. En este caso las imágenes tienen un de tamaño (485, 397, 3) y pasan a ser un vector 1D con $(485 * 397 * 3) = 577,635$ características.

El problema viene con el parámetro *timesteps*, ya que en este caso la longitud de cada secuencia es variable en función del signo, la persona y la mano, pero las redes esperan un intervalo constante. Por ello se han considerado dos opciones: por un lado, emplear una longitud fija de manera que aquellas secuencias que no llegan a este valor se completen con imágenes en negro al final de la secuencia; y, por otro, cargar las secuencias de una en una ($batch = 1$), pero cada una de ellas con la longitud deseada.

Además, se ha encontrado un problema a nivel de memoria al intentar cargar el *dataset* completo. En una de las pruebas de la carga de datos realizada se han redimensionado las imágenes de $(485, 397, 3)$ a $(200, 200, 3)$ y se ha considerado una longitud de secuencia fija de 93 imágenes, correspondiente a la secuencia más larga de todo el conjunto de datos, obteniendo un total de 264,771 imágenes (número de secuencias * 93), lo cual requiere de aproximadamente 200 GB de memoria para cargar. Por este motivo la carga de imágenes se ha realizado implementando un generador de datos (*data generator*).

6.1.1 Generador de datos

La particularidad de los *data generator* reside en que en lugar de cargar el *dataset* entero lo van cargando en tiempo de ejecución a la vez que lo emplean para entrenar la red. De esta manera, cargan un número de imágenes, entrenan la red, liberan la memoria y repiten el proceso.

Hay diferentes maneras de implementar estos generadores, como, por ejemplo, heredando de las clases disponibles en Tensorflow, utilizándolas directamente; o mediante generadores de Python, que es el método empleado en este proyecto.

Al buscar una forma de cargar las secuencias con una longitud variable se ha implementado el generador con esto en mente. De este modo, lo que se hace es leer el fichero con extensión *csv* que contiene la ruta a cada imagen con su correspondiente etiqueta por nombre y secuencia (Figura 6.1). Tras esto se van almacenando las imágenes, previamente normalizadas y concatenadas sus dimensiones en un vector hasta que el número de la secuencia cambia. Cuando esto sucede, este vector de secuencias se añade en un nuevo vector y se devuelve este *array* de tres dimensiones resultante tiene la forma `[númeroSecuencias, imágenesPorSecuencia, características]` donde *númeroSecuencias* es 1, *imágenesPorSecuencia* es la longitud de la secuencia (que en nuestro caso es variable) y *características*, que es un vector de 577,635 elementos con los valores de los píxeles de cada imagen. Esta salida se puede introducir directamente a las redes, ya que es de la misma dimensión que la entrada que esperan `[batch, timestep, feature]`.

label	data
carne_0	/home/paco/saveImg/imagesFixedSize/Paco/alimentos/carne_30.png
carne_0	/home/paco/saveImg/imagesFixedSize/Paco/alimentos/carne_31.png
carne_0	/home/paco/saveImg/imagesFixedSize/Paco/alimentos/carne_32.png
carne_0	/home/paco/saveImg/imagesFixedSize/Paco/alimentos/carne_33.png
carne_0	/home/paco/saveImg/imagesFixedSize/Paco/alimentos/carne_34.png
carne_0	/home/paco/saveImg/imagesFixedSize/Paco/alimentos/carne_35.png
carne_0	/home/paco/saveImg/imagesFixedSize/Paco/alimentos/carne_36.png
carne_0	/home/paco/saveImg/imagesFixedSize/Paco/alimentos/carne_37.png
carne_0	/home/paco/saveImg/imagesFixedSize/Paco/alimentos/carne_38.png
carne_0	/home/paco/saveImg/imagesFixedSize/Paco/alimentos/carne_39.png
carne_0	/home/paco/saveImg/imagesFixedSize/Paco/alimentos/carne_40.png
carne_0	/home/paco/saveImg/imagesFixedSize/Paco/alimentos/carne_41.png
carne_0	/home/paco/saveImg/imagesFixedSize/Paco/alimentos/carne_42.png
carne_0	/home/paco/saveImg/imagesFixedSize/Paco/alimentos/carne_43.png
carne_1	/home/paco/saveImg/imagesFixedSize/Paco/alimentos/carne_44.png
carne_1	/home/paco/saveImg/imagesFixedSize/Paco/alimentos/carne_45.png
carne_1	/home/paco/saveImg/imagesFixedSize/Paco/alimentos/carne_46.png

Figura 6.1: Ejemplo fichero etiquetado *dataset*.

6.1.2 Distribución conjuntos de datos

Las diferentes imágenes se han etiquetado por clase y secuencia de manera individual para cada sujeto y para cada mano, así se tiene un total de 10 ficheros distintos.

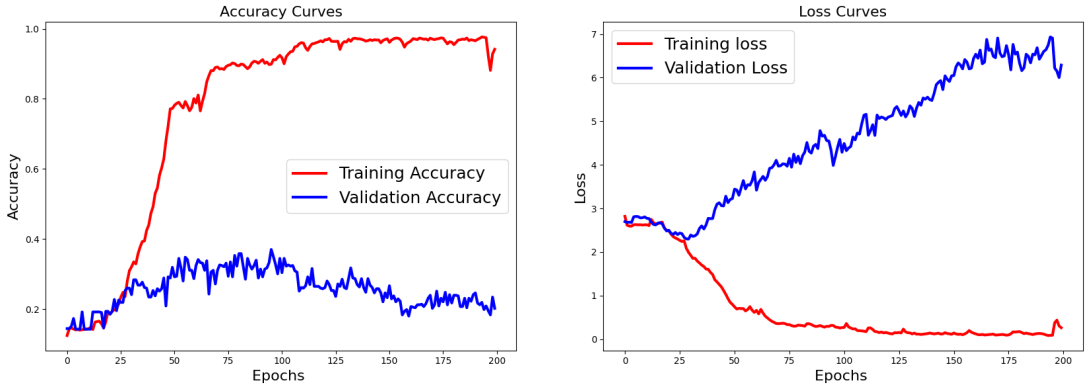
Al distribuirlo de esta manera permite emplear diversas configuraciones para los conjuntos de entrenamiento, validación y test como, por ejemplo, dejar para este último el conjunto completo de una persona que no se haya visto en ninguno de los anteriores.

6.2 Carga del *dataset* completo

Se ha realizado un entrenamiento cargando el conjunto de datos completo en memoria haciendo uso de una arquitectura formada por una capa LSTM compuesta a su vez por 16 unidades de memoria. El modelo se ha entrenado durante 200 épocas con un tiempo medio de 90 segundos por época.

La opción no es viable debido a la gran cantidad de memoria consumida, además, al reducir el tamaño de las imágenes se pierde información, lo que genera que el modelo no se comporte bien. En la Figura 6.2 se pueden observar las gráficas de la pérdida (*loss*) y precisión (*accuracy*) obtenidas con esta arquitectura donde se aprecia como el sistema comienza a sobreentrenar.

En particular, como conjunto de entrenamiento se han empleado 4 personas y 1 para el de validación. Obteniendo el total de datos reflejado en la Tabla 6.1.



(a) Precisión. **(b)** Pérdida.

Figura 6.2: Gráficas carga *dataset* completo.

Signo	N° imágenes	N° secuencias
carne	1,462	131
cenar	1,329	103
comer	1,269	103
cuando	1,146	114
cuchara	1,230	90
cuchillo	1,168	156
desayunar	749	69
donde	1,417	162
gustar	1,311	74
hamburguesa	946	118
huevos	652	72
no gustar	2,018	83
pescado	1,543	76
pizza	1,093	34
plato	528	53
que	1,117	93
querer	1,280	93
sopa	1,105	112
tenedor	1,271	90
tortilla	1,628	226
vaso	1,688	76
verdura	1,531	61
yo	1,205	86

(a) Conjunto de entrenamiento.

Signo	N° imágenes	N° secuencias
carne	352	28
cenar	357	27
comer	454	40
cuando	209	22
cuchara	351	28
cuchillo	316	37
desayunar	172	16
donde	438	52
gustar	323	22
hamburguesa	269	28
huevos	274	25
no gustar	528	23
pescado	320	16
pizza	389	17
plato	242	23
que	286	28
querer	266	21
sopa	349	32
tenedor	314	19
tortilla	413	65
vaso	384	21
verdura	435	15
yo	320	29

(b) Conjunto de validación.

Tabla 6.1: Distribución *dataset*.

6.3 Entrenamiento conjunto de datos completo

Se han realizado diferentes experimentos con el conjunto de datos indicado en la Tabla 6.1 con distintas arquitecturas compuestas por diversas capas tanto LSTM como GRU.

Al cargar el *dataset* mediante un *data generator* en todos los experimentos se conserva el tamaño original de las imágenes (485, 397, 3), evitando así pérdida de información resultante del reescalado.

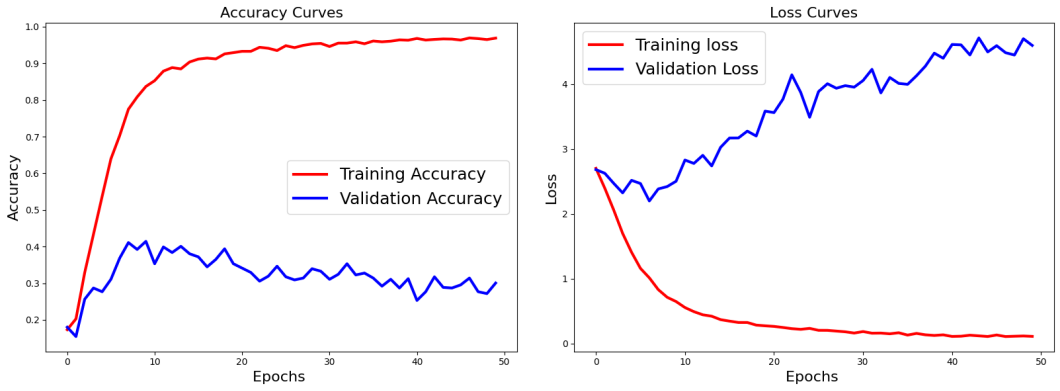
Todos los modelos probados con longitud de secuencia variable tienden a sobreentrenar con relativa facilidad, a partir de las 15-20 épocas de entrenamiento. Por comodidad las etiquetas mostradas en las matrices de confusión se encuentran codificadas como se indica en la Tabla 6.2.

Etiqueta original	**Etiqueta codificada**
carne	0
cenar	1
comer	2
cuando	3
cuchara	4
cuchillo	5
desayunar	6
donde	7
gustar	8
hamburguesa	9
huevos	10
no gustar	11
pescado	12
plato	13
que	14
querer	15
sopa	16
tenedor	17
tortilla	18
vaso	19
verdura	20
yo	21
pizza	22

Tabla 6.2: Codificación etiquetas.

6.3.1 Modelo 1

Arquitectura compuesta por 4 capas GRU donde la primera y la última capas están compuestas por 16 celdas de memoria, mientras que las dos capas intermedias se componen de 8 celdas (Figura 6.8a). Además, a estas capas intermedias se les ha añadido un *dropout*[1] del 25% para intentar corregir el rápido sobreentrenamiento. En la Figura 6.3 se puede observar las gráficas de la pérdida y precisión para un entrenamiento de 50 épocas, donde se aprecia el claro sobreentreniento de la red.



(a) Precisión. **(b)** Pérdida.

Figura 6.3: Gráficas modelo 1.

Al analizar la matriz de confusión de los datos de entrenamiento (Figura 6.4) se ha encontrado que la diagonal principal está prácticamente vacía. Dado que se trata de un modelo sobreentrenado la matriz de confusión del entrenamiento debería salir con todos o casi todos los valores ubicados en la diagonal principal ya que la red ha memorizado estos datos.

Observando la confusión entre las diferentes clases, por ejemplo del signo *comer* (etiqueta 2) con *cenar* (etiqueta 1) y este a su vez con *carne*, se intenta buscar el origen del fallo ya que, tanto en este como en otros casos se trata de signos diferenciables. En varios casos signos bimanuales (*cenar*) son confundidos con otros monomanuales (*comer*).

En los sucesivos modelos probados se ha analizado esta matriz de confusión para comprender de dónde pueden surgir estos resultados. Entre las opciones valoradas se encuentran:

- Elevado número de capas para el problema a resolver.

[1]Desconexión aleatoria de las neuronas al propagar la información.

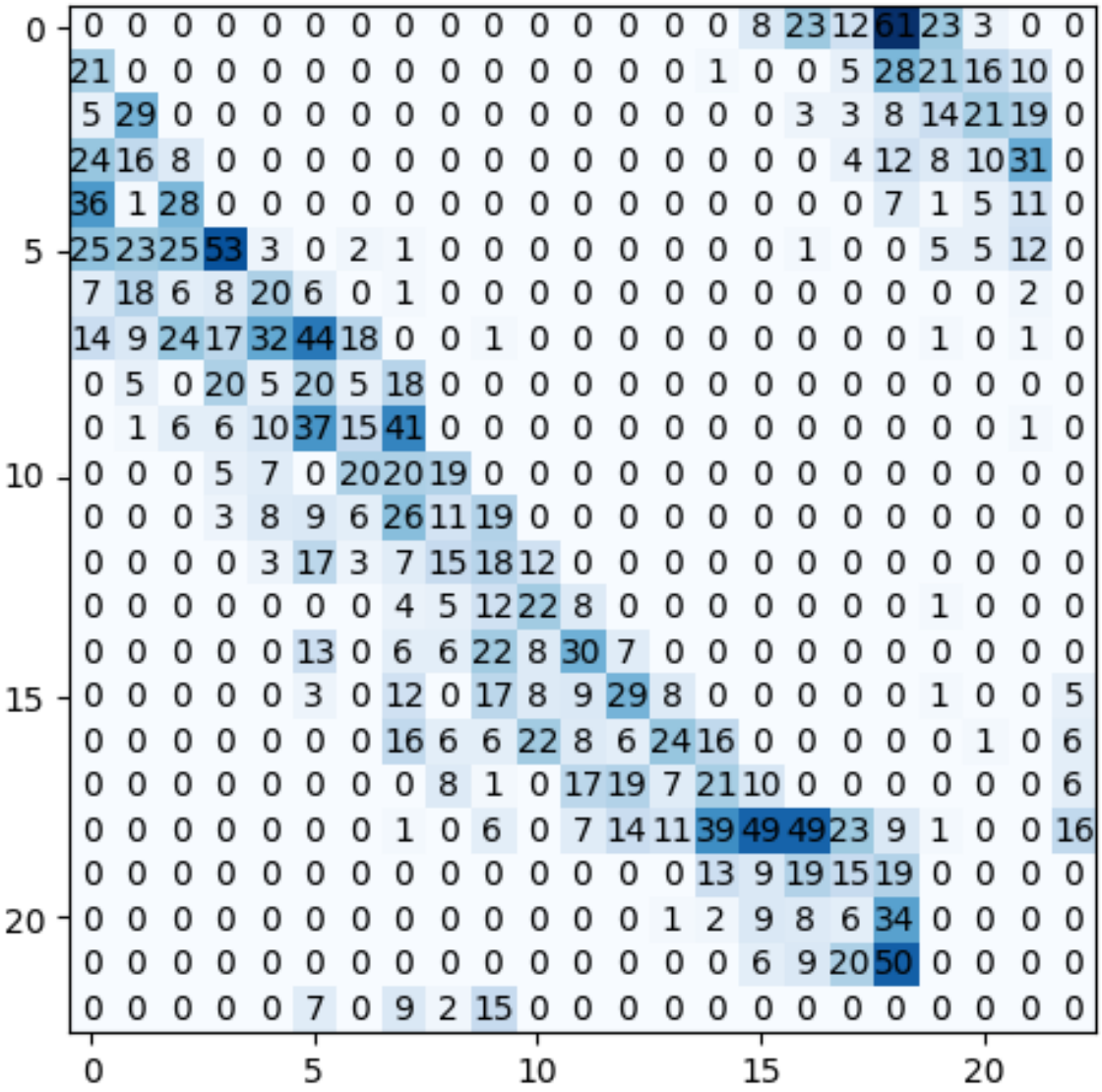


Figura 6.4: Matriz de confusión modelo 1.

- Número de celdas de memoria elevado.
- Alto número de épocas.
- *Dataset* desbalanceado o con demasiado ruido en las secuencias, generando secuencias similares para diferentes etiquetas.

6.3.2 Modelo 2

En este caso se ha empleado una arquitectura compuesta por tres capas LSTM, todas ellas con 16 celdas de memoria (Figura 6.8b). Se han entrenado tres modelos: el primero empleando 50 épocas; el segundo con 20; y, el último con 10. La idea es identificar la relación entre el número de épocas empleado y el resultado obtenido.

El número de épocas influye directamente en el modo que tiene la red de sobreentrenarse, comprobando que con pocas épocas se obtienen mejores resultados. Esto se debe a que llegado un punto la red comienza a aprender características de la imagen que no son únicas del signo y son comunes entre unas secuencias y otras, produciéndose confusiones en la predicción.

6.3.3 Modelo 3

Arquitectura construida por una capa LSTM de 16 unidades seguida de una capa GRU de 8 unidades con un *dropout* del 25% y otra GRU de 16 unidades (Figura 6.8c). Se ha comprobado que emplear una capa intermedia de menor número de unidades junto con el *dropout* retrasa ligeramente el sobreentrenamiento de la red.

Como se ha visto en el modelo previo, el número de épocas influye en los resultados, por lo que en este modelo y los sucesivos se ha optado por un valor inferior a 20 épocas. Para este modelo en particular, el número de épocas utilizado es 20.

No se consigue una mejora apreciable en las gráficas salvo el ligero retraso del sobreentrenamiento.

6.3.4 Modelo 4

Para este modelo se han empleado tres capas GRU la primera y la última compuestas por 16 celdas, mientras que la intermedia mantiene 8 unidades y un *dropout* del 25% (Figura 6.8d).

El número de épocas se ha reducido a 5, consiguiendo de esta manera uno de los resultados más similares a los esperados para un buen entrenamiento en comparación a los modelos anteriores.

En la Figura 6.5 se muestra la matriz de confusión obtenida. En este caso, los valores están más concentrados en la diagonal principal. La Figura 6.6 muestra las gráficas de precisión y pérdida obtenida, donde se puede observar que no se llega a un sobreentramiento completo de la red. Esto indica que los valores fuera de la diagonal principal pueden deberse a la precisión del modelo.

Además, si se analiza esta matriz se puede apreciar que ciertas confusiones son aceptables como, por ejemplo, *comer* (etiqueta 2) y *cuando* (etiqueta 3) ya que ambos consisten en un movimiento del brazo desde la cara hacia delante (la diferencia es la forma de la mano) o los verbos *gustar* (etiqueta 8) y *no gustar* (etiqueta 11) donde el último es una combinación del primero junto con el signo de negación.

6.3.5 Modelo 5

En este modelo se ha optado por emplear una única capa LSTM con diferentes combinaciones del número de unidades para ver la relación de los resultados con éste. Se ha probado un modelo con 16 unidades, otro con 4 unidades y un último con 2, todos ellos durante 10 épocas (Figuras 6.8e, 6.8f y 6.8g respectivamente).

En la Figura 6.7 se muestran las gráficas para los diferentes casos de donde se puede extraer que el número de unidades se encuentra relacionado con la velocidad de sobreentrenamiento de la red. Esto es lógico, ya que la memoria se mantiene durante más tiempo obteniendo mayor cantidad información en el mismo número de épocas y, por tanto, aprendiendo antes.

6.4 Entrenamiento conjunto de datos parcial

La distribución inicial del *dataset* (Tabla 6.1) presenta ruido en algunas de las secuencias, esto es, entre el inicio del signo y el final se encuentran imágenes en las que la mano no se reconoce correctamente. La Figura 6.9 muestra un ejemplo de este tipo de secuencias, mientras que la Figura 6.10 ilustra un ejemplo del mismo signo sin ruido.

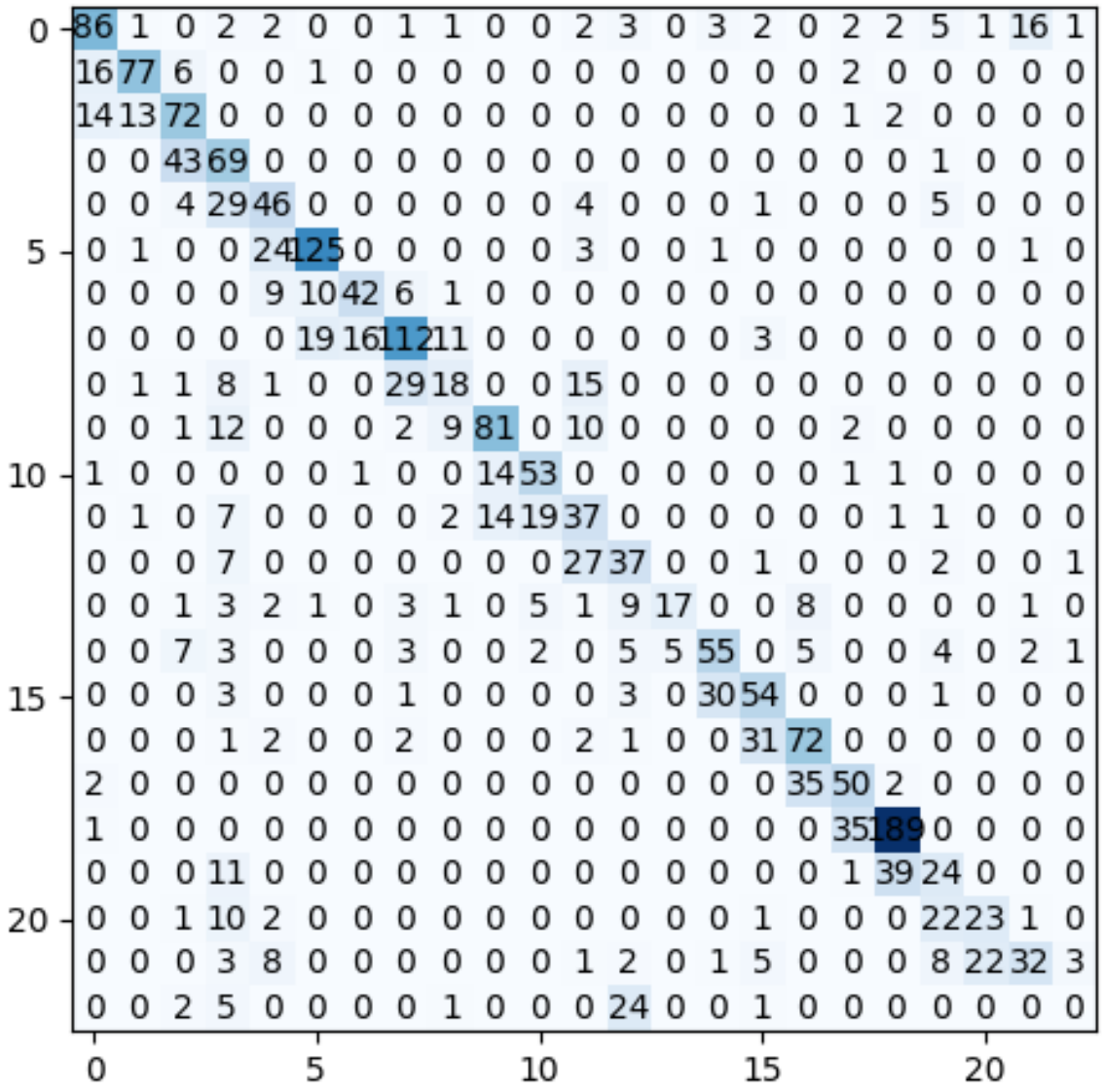


Figura 6.5: Matriz de confusión modelo 4.

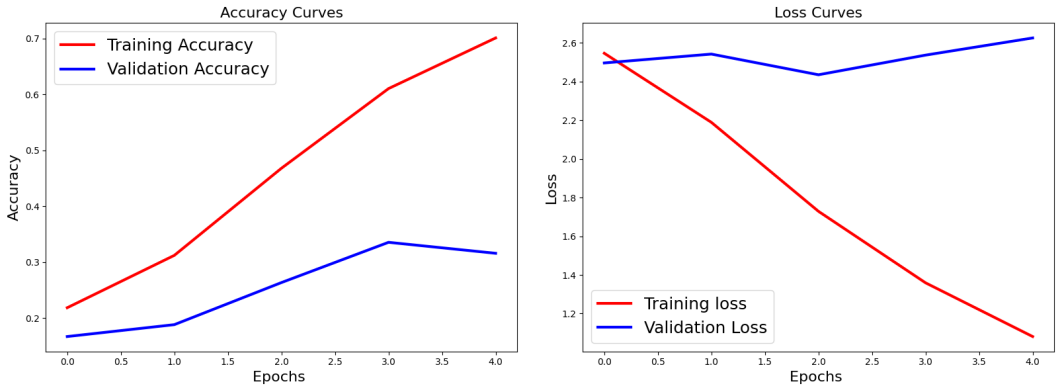


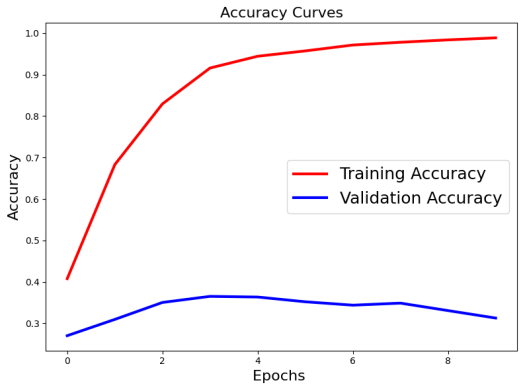
Figura 6.6: Modelo 4.

Con la idea de que el posible mal entrenamiento esté relacionado con el propio conjunto de datos en lugar de con los modelos empleados, se ha creado un nuevo *dataset* a partir del original donde se han eliminado todas estas secuencias con ruido. De los cinco sujetos grabados, la mayor cantidad de ruido se encuentra en dos de ellos, los cuales han sido descartados para este conjunto.

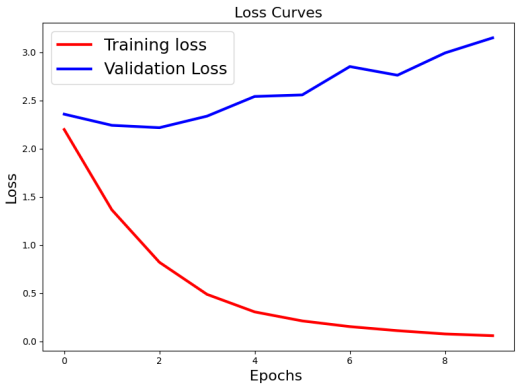
Además, al disminuir el número de secuencias de cada signo no se ha dejado una persona para el conjunto de test y el resto para entrenamiento, sino que se han juntado todos y realizado una división de las secuencias dejando un 80% para entrenamiento y un 20% para test. En la Tabla 6.3 se puede ver la nueva distribución empleada con la que se han probado exactamente los mismos modelos descritos previamente para comprobar las diferencias.

Al emplear este nuevo conjunto de datos, los valores mostrados por las matrices de confusión se concentran más alrededor de la diagonal principal, pero sin conseguir resultados óptimos. De las diferentes arquitecturas destaca el resultado devuelto por el modelo 4 (sección 6.3.4) donde, con esta combinación junto con el nuevo *dataset*, se consiguen resultados dentro de lo *esperado*. La Figura 6.11 muestra la matriz de confusión resultante donde se aprecia la mejora.

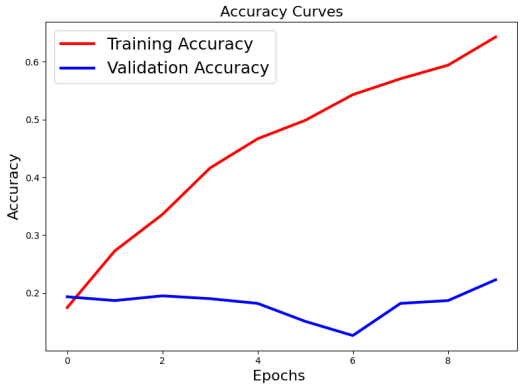
Analizando la matriz en detalle se observa que las confusiones entre las etiquetas suceden todavía en aquellos signos que más relación tienen con la cara, los cuales son los que más ruido introducen en las secuencias, aunque en menor medida a como sucedía con el anterior conjunto de datos.



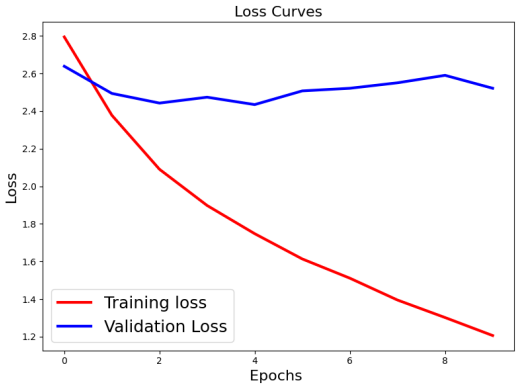
(a) Precisión 16 celdas.



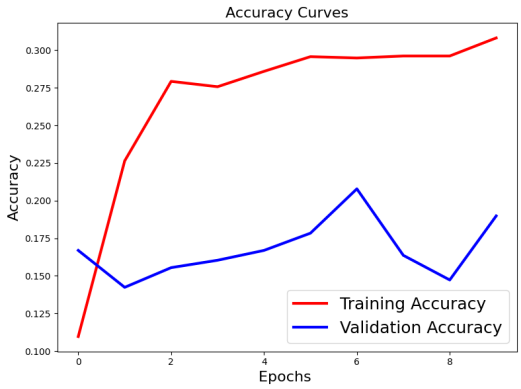
(b) Pérdida 16 celdas.



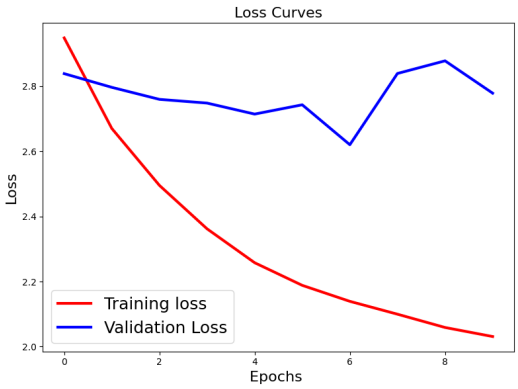
(c) Precisión 4 celdas.



(d) Pérdida 4 celdas.



(e) Precisión 2 celdas.



(f) Pérdida 2 celdas.

Figura 6.7: Comparativa modelo 5.

Layer (type)	Output Shape	Param #
gru (GRU)	(None, None, 16)	27727344
gru_1 (GRU)	(None, None, 8)	624
gru_2 (GRU)	(None, None, 8)	432
gru_3 (GRU)	(None, 16)	1248
dense (Dense)	(None, 23)	391

Total params: 27,730,039
Trainable params: 27,730,039
Non-trainable params: 0

(a) Modelo 1.

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, None, 16)	36969728
lstm_1 (LSTM)	(None, None, 16)	2112
lstm_2 (LSTM)	(None, 16)	2112
dense (Dense)	(None, 23)	391

Total params: 36,974,343
Trainable params: 36,974,343
Non-trainable params: 0

(b) Modelo 2.

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, None, 16)	36969728
gru (GRU)	(None, None, 8)	624
gru_1 (GRU)	(None, 16)	1248
dense (Dense)	(None, 23)	391

Total params: 36,971,991
Trainable params: 36,971,991
Non-trainable params: 0

(c) Modelo 3.

Layer (type)	Output Shape	Param #
gru (GRU)	(None, None, 16)	27727344
gru_1 (GRU)	(None, None, 8)	624
gru_2 (GRU)	(None, 16)	1248
dense (Dense)	(None, 23)	391

Total params: 27,729,607
Trainable params: 27,729,607
Non-trainable params: 0

(d) Modelo 4.

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 16)	36969728
dense (Dense)	(None, 23)	391

Total params: 36,970,119
Trainable params: 36,970,119
Non-trainable params: 0

(e) Modelo 5 (16 celdas).

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 4)	9242240
dense (Dense)	(None, 23)	115

Total params: 9,242,355
Trainable params: 9,242,355
Non-trainable params: 0

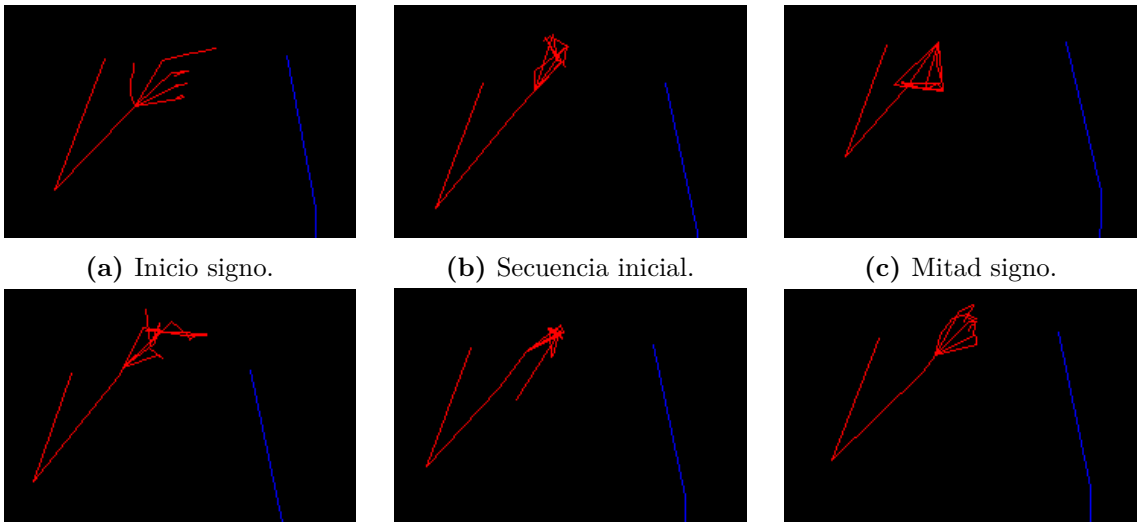
(f) Modelo 5 (4 celdas).

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 2)	4621104
dense (Dense)	(None, 23)	69

Total params: 4,621,173
Trainable params: 4,621,173
Non-trainable params: 0

(g) Modelo 5 (2 celdas).

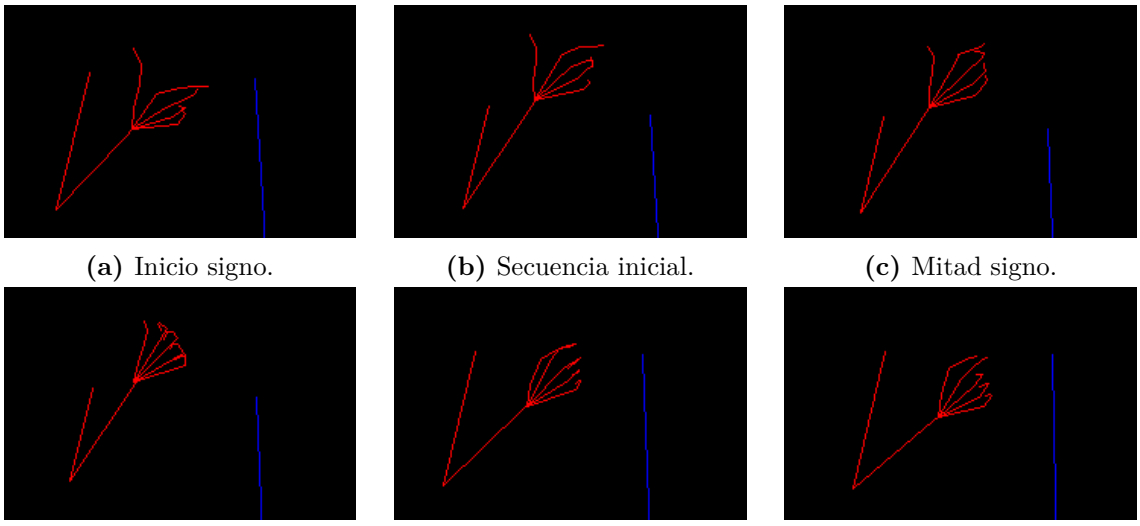
Figura 6.8: Arquitecturas modelos empleados.



(a) Inicio signo. **(b)** Secuencia inicial. **(c)** Mitad signo.

(d) Secuencia final. **(e)** Secuencia final. **(f)** Fin signo.

Figura 6.9: Secuencia con ruido.



(a) Inicio signo. **(b)** Secuencia inicial. **(c)** Mitad signo.

(d) Secuencia final. **(e)** Secuencia final. **(f)** Fin signo.

Figura 6.10: Secuencia sin ruido.

Signo	N° imágenes	N° secuencias
carne	954	81
cenar	1,076	73
comer	1,198	89
cuando	860	72
cuchara	1,072	69
cuchillo	986	125
desayunar	554	46
donde	1,105	122
gustar	980	54
hamburguesa	807	89
huevos	547	49
no gustar	1,550	61
pescado	1,109	52
pizza	1,058	32
plato	394	35
que	1,025	72
querer	920	61
sopa	889	80
tenedor	1,001	58
tortilla	1,219	177
vaso	1,271	57
verdura	1,326	48
yo	1,011	70

(a) Conjunto de entrenamiento.

Signo	N° imágenes	N° secuencias
carne	499	31
cenar	284	21
comer	275	24
cuando	186	19
cuchara	210	18
cuchillo	277	34
desayunar	151	12
donde	281	33
gustar	282	14
hamburguesa	181	23
huevos	139	13
no gustar	414	16
pescado	335	12
pizza	212	8
plato	82	9
que	231	17
querer	245	16
sopa	238	21
tenedor	242	15
tortilla	291	45
vaso	353	15
verdura	280	12
yo	270	18

(b) Conjunto de test.

Tabla 6.3: Distribución nuevo *dataset*.

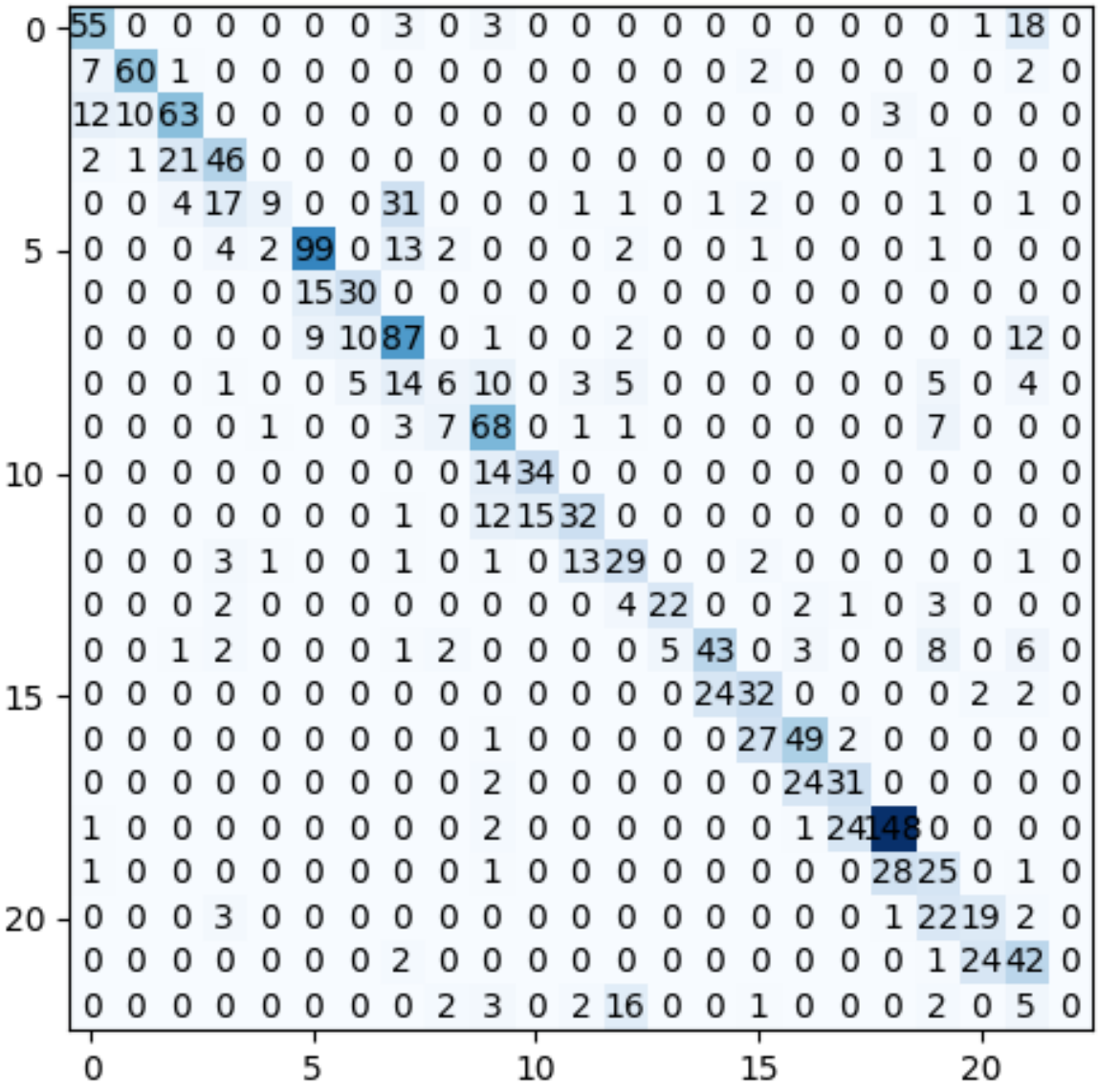


Figura 6.11: Matriz de confusión modelo 4 nuevo *dataset*.

6.5 Entrenamiento secuencia fija

Se ha realizado también un experimento con valores de secuencias fijos para todas las imágenes, aunque eso suponga la introducción de ruido en gran parte de éstas. Para ello, se ha buscado la secuencia con mayor longitud de todo el *dataset*. En nuestro caso, la mayor secuencia está compuesta por 93 imágenes, valor que se ha establecido como longitud por defecto para todas las demás secuencias. De esta manera, las secuencias que no llegan a tal longitud son complementadas con imágenes en negro al final.

El entrenamiento se ha realizado usando el modelo 4 (sección 6.3.4), ya que es el que mejores resultados ha devuelto en los casos anteriores, empleando el conjunto de datos completo y 25 épocas.

Uno de los principales inconvenientes encontrados se encuentra en el tiempo necesario para el entrenamiento al aumentar considerablemente el número de imágenes. Este tiempo se sitúa sobre los 3,750 segundos por época, consiguiendo el entrenamiento completo en aproximadamente 26 horas.

En la Figura 6.12 se muestran las gráficas de precisión y pérdida obtenidas. Se puede observar cómo el modelo requiere de un mayor número de épocas para llegar a sobreentrenarse, debido a este incremento de imágenes.

La Figura 6.13 muestra la matriz de confusión obtenida para el conjunto de entrenamiento donde se puede observar que aunque algunos datos se mantienen a lo largo de la diagonal principal, gran parte de ellos se desvían, lo cual indica una mala predicción por parte del modelo.

Analizando en detalle la matriz se observa que los signos más largos como *verdura* (etiqueta 20) o *sopa* (etiqueta 16) son los que tienen una tasa de acierto elevada ya que la introducción de ruido es menor. En cambio, aquellos con las secuencias más cortas y que sufren una introducción de ruido mayor como *hamburguesa* (etiqueta 9) y *cuchillo* (etiqueta 5) son confundidos.

Por lo tanto se puede concluir que el modelo es incapaz de distinguir correctamente en todos los casos debido a estas nuevas imágenes en negro introducidas al final de cada secuencia. Esto se debe a que en las secuencias cortas puede haber más cantidad de ruido que de datos, haciendo que dos clases inicialmente distintas parezcan la misma. En el caso de las secuencias

largas, al introducir una cantidad pequeña de ruido no es tan notable y la predicción sucede dentro de lo esperado.

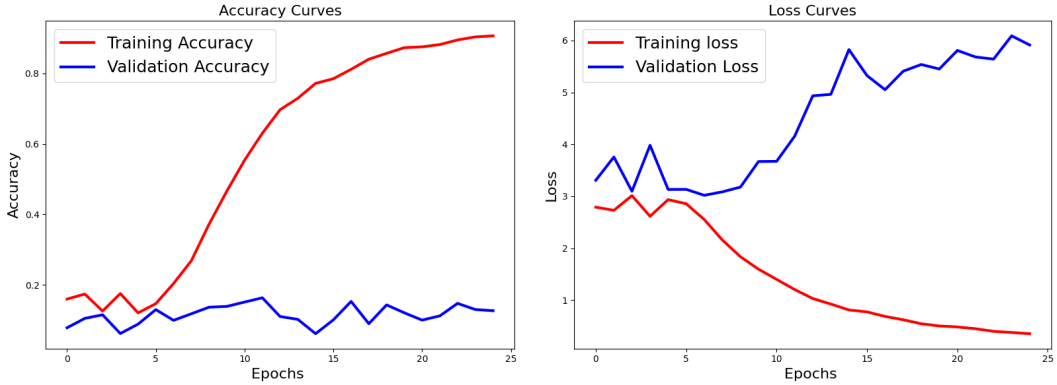


Figura 6.12: Gráficas modelo de secuencias fijas.

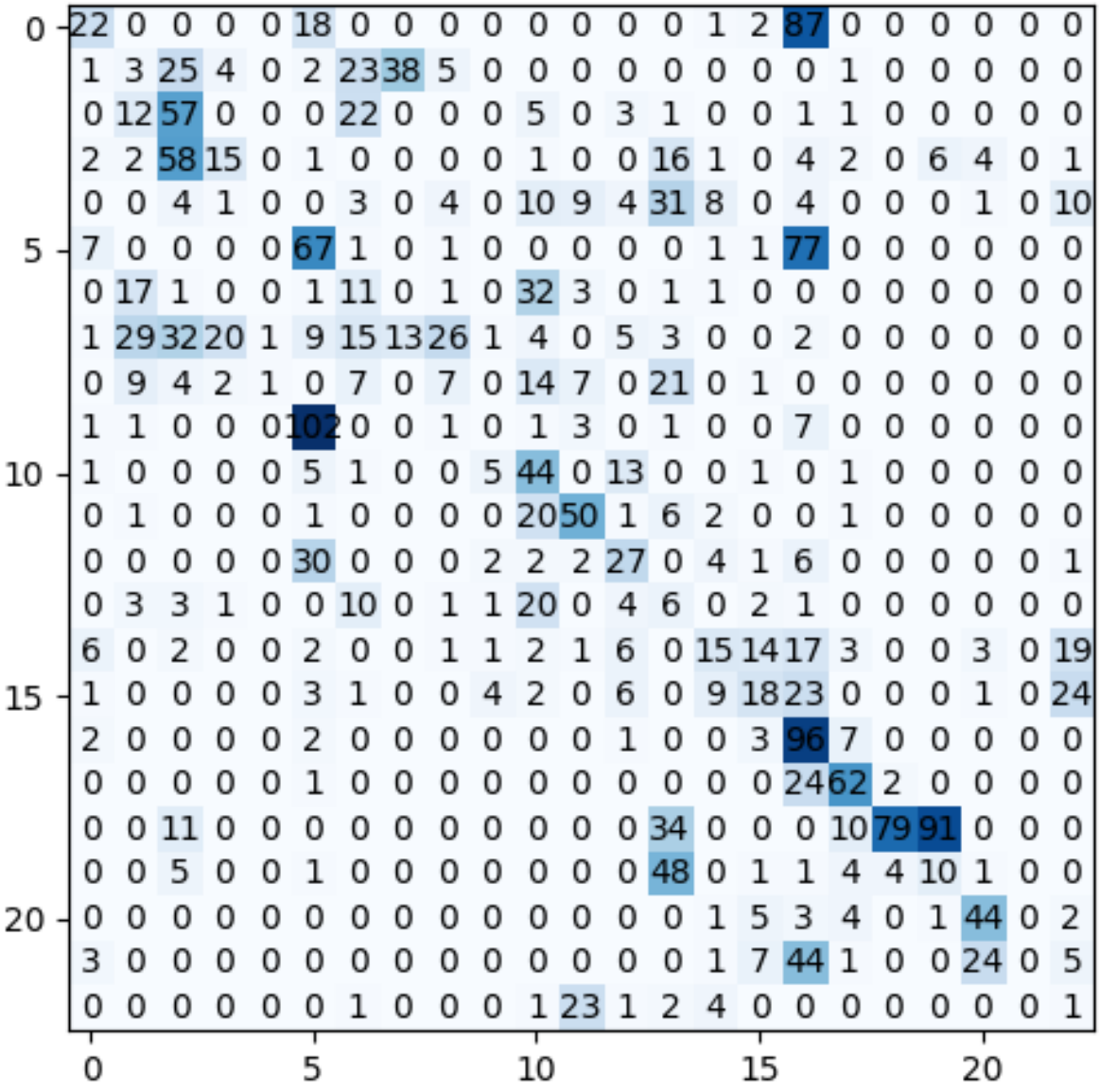


Figura 6.13: Matriz de confusión modelo de secuencias fijas.

7 Conclusiones

De los diferentes objetivos planteados al inicio de este TFM se ha conseguido alcanzar una gran parte de manera satisfactoria, como son el correcto control de la cámara Azure Kinect y su uso para el reconocimiento de signos o la creación de un *dataset* correspondiente a un vocabulario reducido de la LSE para una interacción básica sobre la tarea diaria de comer, inexistente hasta el momento.

Sin embargo, respecto al objetivo principal consistente en crear un intérprete automático para ciertas palabras o frases, el resultado no ha sido el esperado.

En lugar de desarrollar un sistema funcional, se ha realizado una comparativa de diversas arquitecturas de RNN y con diferentes distribuciones en los datos para ver la influencia de éstos sobre los resultados obtenidos. Este estudio comparativo muestra que el origen del reconocimiento pobre que se ha obtenido durante el desarrollo de este TFM se encuentra tanto en unos datos iniciales compuestos por ruido en ciertas secuencias, lo que impide su correcta identificación; así como en unos modelos cuyo sobreentrenamiento es demasiado rápido, ligado también a los datos ya que, en términos generales, las secuencias son relativamente cortas.

Ante estas adversidades, se plantea como trabajo futuro la corrección de estas secuencias, además de la búsqueda de otras arquitecturas enfocadas a longitudes de secuencias variables.

Bibliografía

[1] Página oficial Azure Kinect. https://azure.microsoft.com/es-es/services/kinect-dk/. Accedido el 28-06-2021.

[2] Azure Kinect body tracking index map. https://docs.microsoft.com/en-us/azure/kinect-dk/body-index-map. Accedido el 28-06-2021.

[3] Azure Kinect DK coordinate systems. https://docs.microsoft.com/en-us/azure/kinect-dk/coordinate-systems. Accedido el 28-06-2021.

[4] Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Realtime multi-person 2d pose estimation using part addinity fields. *arXiv:1611.08050*, 2017.

[5] Tomas Simon, Hanbyul Joo, Iain Matthews, and Yaser Sheikh. Hand keypoint detection in single images using multiview bootstrapping. *arXiv:1704.07809v1*, 2017.

[6] Jordi TORRRES.AI. https://torres.ai/redes-neuronales-recurrentes/. Accedido el 28-06-2021.

[7] Especificaciones hardware Azure Kinect DK. https://docs.microsoft.com/en-us/azure/kinect-dk/hardware-specification. Accedido el 28-06-2021.

[8] Organización mundial de la Salud. https://www.who.int. Accedido el 15-05-2021.

[9] M.B. Waldron and Soowon Kim. Isolated ASL sign recognition system for deaf persons. In *IEEE Transactions on Rehabilitation Engineering*, volume 3, pages 261–271, 1995.

[10] Mohammed Waleed Kadous. Machine recognition of auslan signs using PowerGloves: Towards large-lexicon recognition of sign language. In *Proceedings of the Workshop on the Integration of Gesture in Language and Speech*, pages 165–174, 1996.

[11] Christian Vogler and Dimitris Metaxas. Adapting Hidden Markov Models for ASL recognition by using three-dimensional computer vision methods. In *In Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, pages 156–161, 1997.

[12] J. Starner, J. Weaver, and A. Pentland. Real-time American Sign Language recognition using desk and wearable computer based video. In *IEEE Transactions on Pattern Analysis and Machine Inteligence*, volume 20, pages 1371–1375, 1998.

[13] Kai Nickel and Rainer Stiefelhagen. Visual recognition of pointing gestures for human-robot. *Image Vision Comput.*, 25:1875–1884, 12 2007.

[14] Eun-Jung Holden and Robyn Owens. Visual sign language recognition. In *Multi-Image Analysis*, volume 2032, pages 270–287, 01 2001.

[15] J. Han, George Awad, and A. Sutherland. Automatic skin segmentation and tracking in sign language recognition. *Computer Vision, IET*, 3:24 – 35, 04 2009.

[16] Lionel Pigou, Sander Dieleman, Pieter-Jan Kindermans, and Benjamin Schrauwen. Sign language recognition using convolutional neural networks. In *Computer Vision - ECCV 2014 Workshops: Zurich, Switzerland*, volume 8925, pages 572–578, 03 2015.

[17] Chalearn looking at people @ eccv2014. `http://gesture.chalearn.org/2014-looking-at-people-challenge`, 2014. Accessed: 2nd July 2020.

[18] Deepasl: Enabling ubiquitous and non-intrusive word and sentence-level sign language translation. *CoRR*, abs/1802.07584, 2018. Withdrawn.

[19] Leap Motion Controller. `https://www.ultraleap.com/product/leap-motion-controller/`. Accedido el 28-06-2021.

[20] Thad Starner and Alex Pentland. Real-time american sign language recognition from video using hidden markov models. In *Computational Imaging and Vision*, pages 227–243. Springer Netherlands, 1997.

[21] API Azure Kinect Sensor SDK. `https://microsoft.github.io/Azure-Kinect-Sensor-SDK/master/index.html`. Accedido el 28-06-2021.

[22] Azure Kinect body tracking joints. `https://docs.microsoft.com/en-us/azure/kinect-dk/body-joints`. Accedido el 28-06-2021.

[23] Zhe Cao, Gines Hidalgo, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Openpose: Realtime multi-person 2D pose estimation using part affinity fields. *arXiv:1812.08008v2*, 2019.

[24] Proyecto pyk4a en GitHub por etiennedub. `https://github.com/etiennedub/pyk4a`. Accedido el 28-06-2021.

[25] Proyecto Azure-Kinect-Python en GitHub por hexops. `https://github.com/hexops/Azure-Kinect-Python`. Accedido el 28-06-2021.

[26] Proyecto pyKinectAzure en GitHub por ibaiGorordo. `https://github.com/ibaiGorordo/pyKinectAzure`. Accedido el 28-06-2021.

[27] Página del proyecto Openpose. `https://github.com/CMU-Perceptual-Computing-Lab/openpose`. Accedido el 26-05-2021.

[28] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[29] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, 2013.

[30] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[31] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches, 2014.

[32] Rafael Muñoz-Salinas, R. Medina-Carnicer, F.J. Madrid-Cuevas, and A. Carmona-Poyato. Depth silhouettes for gesture recognition. In *Pattern Recognition Letters*, volume 29, pages 319–329, 2008.

[33] S. Liwicki and M. Everingham. Automatic recognition of fingerspelled words in british sign language. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 50–57, 2009.

[34] Max Kuhn and Kjell Johnson. *Applied Predictive Modeling*, page 70. 2013.

[35] Tabla de absorción espectro infrarrojo. `https://www.sigmaaldrich.com/ES/es/technical-documents/technical-article/analytical-chemistry/photometry-and-reflectometry/ir-spectrum-table`. Accedido el 28-06-2021.

Lista de Acrónimos y Abreviaturas

ASL	Lengua de Signos Americana.
CNN	Redes Neuronales Convolucionales.
CTC	Clasificación Temporal Conexionista.
FPS	Frames Por Segundo.
GRU	Gated Recurring Units.
HMM	Modelo Oculto de Markov.
LSE	Lengua de Signos Española.
LSTM	Long Short-Term Memory.
NFoV	*Narrow Field of View.*
OMS	Organización Mundial de la Salud.
ReLUs	Unidades de Rectificado Lineal.
RNA	Redes Neuronales Artificiales.
RNN	Red Neuronal Recurrente.
RNNs	Redes Neuronales Recurrentes.
SO	Sistema Operativo.
TFM	Trabajo Final de Máster.
ToF	*Time of Flight.*
WFoV	*Wide Field of View.*